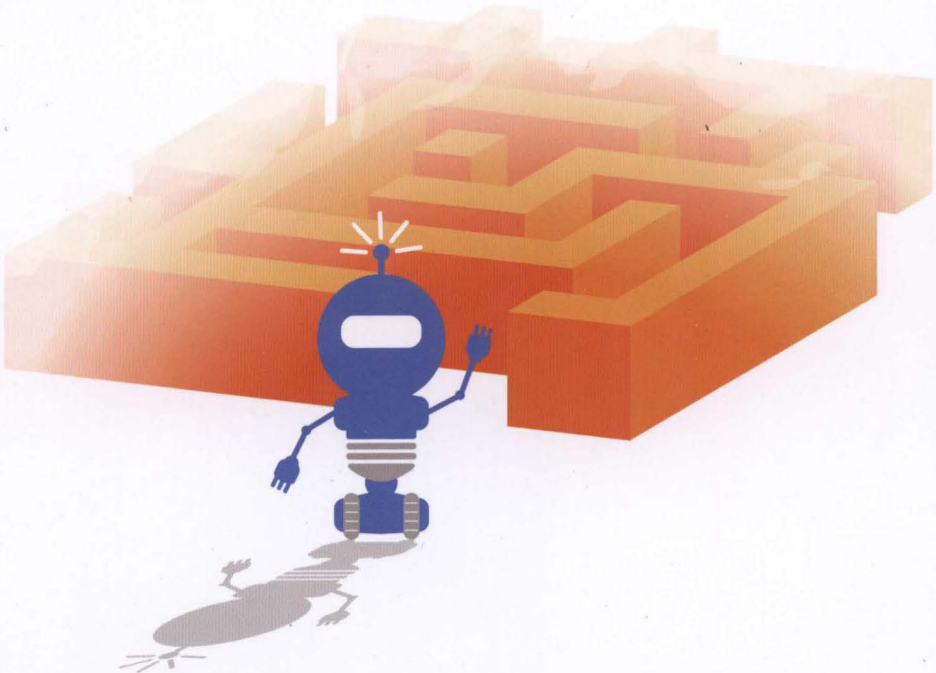
版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

Broadview®
www.broadview.com.cn

博文视点AI系列

从原理剖析到算法精髓，再到TensorFlow代码实现
掌握用强化学习思想解决实际问题的能力
配书源代码供下载



强化学习精要

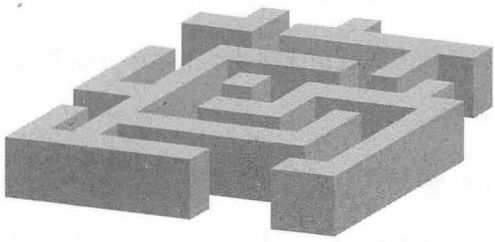
核心算法与TensorFlow实现

冯超◎著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

仅供非商业用途或交流学习使用

博文视点AI系列



强化学习精要

核心算法与TensorFlow实现

冯超◎著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

内容简介

《强化学习精要：核心算法与 TensorFlow 实现》用通俗幽默的语言深入浅出地介绍了强化学习的基本算法与代码实现，为读者构建了一个完整的强化学习知识体系，同时介绍了这些算法的具体实现方式。从基本的马尔可夫决策过程，到各种复杂的强化学习算法，读者都可以从本书中学习到。本书除了介绍这些算法的原理，还深入分析了算法之间的内在联系，可以帮助读者举一反三，掌握算法精髓。书中介绍的代码可以帮助读者快速将算法应用到实践中。

《强化学习精要：核心算法与 TensorFlow 实现》内容翔实，语言简洁易懂，既适合零基础的人员入门学习，也适合相关科研人员研究参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

强化学习精要：核心算法与 TensorFlow 实现 / 冯超著.—北京：电子工业出版社，2018.6
（博文视点 AI 系列）
ISBN 978-7-121-34000-0
I. ①强…II. ①冯…III. ①机器学习—算法—研究 IV. ①TP181
中国版本图书馆 CIP 数据核字（2018）第 068910 号

策划编辑：郑柳洁
责任编辑：郑柳洁
印　　刷：天津嘉恒印务有限公司
装　　订：天津嘉恒印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：720×1000　1/16　　印张：24　字数：470 千字
版　　次：2018 年 6 月第 1 版
印　　次：2018 年 6 月第 1 次印刷
定　　价：80.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

前言

从你拿起本书的那一刻起，我们就一起踏上了这段有关强化学习的冒险之旅。本书中有简单直白的叙述，也有复杂冗长的证明；有诙谐幽默，也有深刻的思考。所有的一切，都是为了帮助你更轻松地对强化学习有更多了解、更多感悟。

一年之前，我在拙作《深度学习轻松学：核心算法与视觉实践》中，同样以这段话作为开篇，唯一不同的是这一次书的主题换成了强化学习。近年来，强化学习领域有了很大的发展，基于强化学习开发的智能体也被人们熟知。于是越来越多的人开始关注强化学习，了解这个领域，并使用强化学习的方法解决问题。随着深度学习的发展，强化学习也逐渐发挥出了自己的实力，凭借更灵活的问题定义方式，解决了很多其他方法难以解决的问题。

虽然强化学习这个领域已经有了多年的积累与发展，但是与深度学习相比，国内外与此相关的书籍与课程偏少。同时，由于强化学习融合了很多领域的知识，本身的技术难度较大，学习起来比较困难。两者叠加，使得强化学习仍然是一个令很多人感到困惑的概念。

为了学习强化学习中的各种知识，作为本书的作者，我也花费了很多的精力。在阅读了很多经典的书籍、论文和博客之后，我开始对强化学习有了自己的理解和体会。同时也积累了不少关于强化学习的“学习笔记”，其中的一部分已经在一些公开场合，例如我的知乎专栏《无痛的机器学习》（https://zhuanlan.zhihu.com/hsmyy）中与读者分享过。在交流的过程中，我能感受到读者对强化学习的浓厚兴趣，同时自己也在分享中得到了提高。

经过长时间的积累，这些心得与体会最终凝结成了本书。在这个过程中我重新回顾了自己所学的知识，也进一步思考了强化学习各部分知识点之间的联系。我希望可以给大家呈现一个尽可能完整而丰富的强化学习世界，与大家共同探讨强化学习中的无限可能。

为了使更多的人能够从本书中得到收获，我选择强化学习中比较经典的算法进行了介绍，这些都是强化学习中十分重要的内容，对不了解强化学习的读者来说，可以帮助你快速了解强化学习的知识体系，对这个领域有一个全面的认识；对正在学习强化

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

学习的读者来说，本书中的知识同样可以帮助你梳理强化学习的核心要点，加深对这些基本问题的理解。

在编写这本书的过程中，我又一次体会到求知的艰辛。获取知识的道路总是充满荆棘，除了自身不断地努力，更少不了身边人对我的支持鼓励。感谢家人对我的理解与支持，感谢编辑郑柳洁女士再次成为我的伯乐，为我提供了很多帮助，感谢刘晓鸣先生为本书绘制了大量插图。我相信我们的努力付出会使更多的人享受获得知识的快乐。由于本人才疏学浅，行文间难免有所纰漏，望各位读者多多包涵，不吝赐教。

作　者

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书提供示例代码资源文件，可在【下载资源】处下载。
- **提交勘误**：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与读者交流**：在页面下方【读者评论】处留下您的疑问或观点，与其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34000



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

目录

第一部分　强化学习入门与基础知识

1　引言　2

1.1　强化学习的概念　2

1.1.1　巴浦洛夫的狗　3

1.1.2　俄罗斯方块　4

1.2　站在被实验者的角度看问题　5

1.3　强化学习效果的评估　8

1.3.1　不断试错　8

1.3.2　看重长期回报　8

1.4　强化学习与监督学习　9

1.4.1　强化学习与监督学习的本质　9

1.4.2　模仿学习　10

1.5　强化学习的实验环境　11

1.5.1　Arcade Learning Environment　12

1.5.2　Box2D　12

1.5.3　MuJoCo　13

1.5.4　Gym　14

1.6　本书的主要内容　15

1.7　参考资料　16

2　数学与机器学习基础　17

2.1　线性代数基础　17

2.2　对称矩阵的性质　21

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2.2.1 特征值与特征向量 21
2.2.2 对称矩阵的特征值和特征向量 22
2.2.3 对称矩阵的对角化 23
2.3 概率论 24
2.3.1 概率与分布 24
2.3.2 最大似然估计 27
2.4 重要性采样 29
2.5 信息论基础 33
2.6 KL 散度 35
2.7 凸函数及其性质 39
2.8 机器学习的基本概念 41
2.9 机器学习的目标函数 43
2.10 总结 45

3 优化算法 47

3.1 梯度下降法 47
3.1.1 什么是梯度下降法 47
3.1.2 优雅的步长 48
3.2 动量算法 53
3.3 共轭梯度法 59
3.3.1 精妙的约束 59
3.3.2 共轭 60
3.3.3 优化步长的确定 63
3.3.4 Gram-Schmidt 方法 64
3.3.5 共轭梯度 65
3.4 自然梯度法 69
3.4.1 基本概念 69
3.4.2 Fisher 信息矩阵 71
3.4.3 自然梯度法目标公式 76
3.5 总结 77

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

4 TensorFlow 入门 78

4.1 TensorFlow 的基本使用方法 78

4.2 TensorFlow 原理介绍 82

4.2.1 创建变量的 scope 83

4.2.2 创建一个 Variable 背后的故事 89

4.2.3 运算操作 94

4.2.4 tf.gradients 96

4.2.5 Optimizer 102

4.2.6 TensorFlow 的反向传播技巧 106

4.2.7 arg_scope 的使用 109

4.3 TensorFlow 的分布式训练 113

4.3.1 基于 MPI 的数据并行模型 114

4.3.2 MPI 的实现：mpi_adam 121

4.4 基于 TensorFlow 实现经典网络结构 122

4.4.1 多层感知器 122

4.4.2 卷积神经网络 124

4.4.3 循环神经网络 126

4.5 总结 129

4.6 参考资料 129

5 Gym 与 Baselines 130

5.1 Gym 130

5.1.1 Gym 的安装 130

5.1.2 Gym 的基本使用方法 132

5.1.3 利用 Gym 框架实现一个经典的棋类游戏：蛇棋 134

5.2 Baselines 138

5.2.1 Baselines 中的 Python 3 新特性 139

5.2.2 tf_util 141

5.2.3 对 Gym 平台的扩展 142

5.3 总结 144

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

6 强化学习基本算法 145

6.1 马尔可夫决策过程 145

6.1.1 MDP：策略与环境模型 145

6.1.2 值函数与 Bellman 公式 147

6.1.3 “表格式”Agent 151

6.2 策略迭代 153

6.2.1 策略迭代法 153

6.2.2 策略提升的证明 159

6.2.3 策略迭代的效果展示 160

6.3 价值迭代 162

6.3.1 N 轮策略迭代 162

6.3.2 从动态规划的角度谈价值迭代 165

6.3.3 价值迭代的实现 167

6.4 泛化迭代 168

6.4.1 两个极端 168

6.4.2 广义策略迭代法 169

6.4.3 泛化迭代的实现 170

6.5 总结 171

第二部分 最优价值算法

7 Q-Learning 基础 173

7.1 状态转移概率：从掌握到放弃 173

7.2 蒙特卡罗方法 174

7.3 探索与利用 178

7.4 蒙特卡罗的方差问题 181

7.5 时序差分法与 SARSA 183

7.6 Q-Learning 186

7.7 Q-Learning 的收敛性分析 189

7.8 从表格形式到价值模型 193

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

7.9 Deep Q Network 195
7.10 总结 202
7.11 参考资料 202

8 DQN 的改进算法 203

8.1 Double Q-Learning 203
8.2 Priority Replay Buffer 204
8.3 Dueling DQN 209
8.4 解决 DQN 的冷启动问题 211
8.5 Distributional DQN 214
8.5.1 输出价值分布 214
8.5.2 分布的更新 216
8.6 Noisy Network 218
8.7 Rainbow 221
8.7.1 Rainbow 的模型特点 221
8.7.2 Deep Q Network 的实现 223
8.8 总结 227
8.9 参考资料 227

第三部分 基于策略梯度的算法

9 基于策略梯度的算法 229

9.1 策略梯度法 229
9.1.1 算法推导 230
9.1.2 算法分析 233
9.1.3 算法改进 234
9.2 Actor-Critic 算法 236
9.2.1 降低算法的方差 236
9.2.2 A3C 算法 238
9.2.3 A2C 算法实战 240

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

9.3 总结 243
9.4 参考资料 243

10 使策略单调提升的优化算法 244

10.1 TRPO 244
10.1.1 策略的差距 245
10.1.2 策略提升的目标公式 247
10.1.3 TRPO 的目标定义 248
10.1.4 自然梯度法求解 251
10.1.5 TRPO 的实现 254
10.2 GAE 256
10.2.1 GAE 的公式定义 256
10.2.2 基于 GAE 和 TRPO 的值函数优化 259
10.2.3 GAE 的实现 260
10.3 PPO 261
10.3.1 PPO 介绍 261
10.3.2 PPO 算法实践 263
10.4 总结 264
10.5 参考资料 264

11 Off-Policy 策略梯度法 265

11.1 Retrace 266
11.1.1 Retrace 的基本概念 266
11.1.2 Retrace 的算法实现 267
11.2 ACER 270
11.2.1 Off-Policy Actor-Critic 270
11.2.2 ACER 算法 272
11.2.3 ACER 的实现 276
11.3 DPG 279
11.3.1 连续空间的策略优化 279

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

11.3.2 策略模型参数的一致性 280
11.3.3 DDPG 算法 283
11.3.4 DDPG 的实现 286
11.4 总结 289
11.5 参考资料 289

第四部分 其他强化学习算法

12 稀疏回报的求解方法 291
12.1 稀疏回报的困难 291
12.2 层次强化学习 294
12.3 HER 298
12.3.1 渐进式学习 299
12.3.2 HER 的实现 301
12.4 总结 304
12.5 参考资料 304

13 Model-based 方法 305
13.1 AlphaZero 305
13.1.1 围棋游戏 305
13.1.2 Alpha-Beta 树 307
13.1.3 MCTS 309
13.1.4 策略价值模型 312
13.1.5 模型的对决 316
13.2 iLQR 316
13.2.1 线性模型的求解法 317
13.2.2 非线性模型的解法 322
13.2.3 iLQR 的实现 325
13.3 总结 328
13.4 参考资料 328

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

第五部分　反向强化学习

14　反向强化学习入门　330

14.1 基本概念 330
14.2 从最优策略求解回报 332
14.2.1 求解回报的目标函数 332
14.2.2 目标函数的约束 334
14.3 求解线性规划 335
14.3.1 线性规划的求解过程 335
14.3.2 实际案例 337
14.4 无限状态下的求解 338
14.5 从样本中学习 342
14.6 总结 344
14.7 参考资料 344

15　反向强化学习算法 2.0　345

15.1 最大熵模型 345
15.1.1 指数家族 346
15.1.2 最大熵模型的推导 349
15.1.3 最大熵模型的实现 354
15.2 最大熵反向强化学习 356
15.3 GAIL 361
15.3.1 GAN 的基本概念 361
15.3.2 GAN 的训练分析 363
15.4 GAIL 实现 367
15.5 总结 370
15.6 参考资料 370

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

第一部分
强化学习入门与基础知识

作为本书的第一部分，我们首先介绍强化学习的基本概念和基本框架，同时将与强化学习有关的所有基础知识做详细介绍。虽然这一部分并没有讲述太多强化学习的内容，但它仍然十分重要，只有了解了这些内容，才能更好地学习后面的知识。如果读者已经对数学基础和机器学习的基本概念有一定的了解，可以通过阅读本部分复习；如果读者对上述内容不太了解，则要认真学习这一部分的知识。

第一部分共包含六章：

- 第 1 章介绍了强化学习的定义、目标等基础知识，并介绍了全书的结构。
- 第 2 章介绍了数学基础知识。
- 第 3 章介绍了本书使用的优化算法相关的知识。
- 第 4 章介绍了本书代码使用的框架——TensorFlow 的基本知识。
- 第 5 章介绍了强化学习相关的项目——Gym 和 Baselines 的基本知识。
- 第 6 章介绍了强化学习的理论基础——马尔可夫决策过程及基于动态规划的求解方法。

仅供非商业用途或交流学习使用



引言

1.1 强化学习的概念

作为开篇，最重要的一件事是介绍强化学习的入门知识和相关应用。我们首先要解决一个问题：什么是强化学习？这个词并不那么容易理解，“强化”在这里是一个动词还是一个名词，又或者是一个人名？还有人把它称为“增强学习”，其实名字所表达的含义差不多，但总感觉这个名字并没有很清楚地表达它想表达的含义，即使一些对强化学习有一定经验的人也可能感到疑惑。

不得不承认，当今很多知识和概念都是从西方传来的，强化学习也是一个“外来品”。它的英文是 Reinforcement Learning，在英文辞典里查询 reinforcement 这个词，我们找到了下面的解释（来自https://www.merriam-webster.com/dictionary/reinforcement）：

> something that strengthens or encourages something, such as a response to someone's behavior that is intended to make that person more likely to behave that way again.

将英文翻译后的大概意思是：“reinforcement”是一种“套路”，或者是某种方法，也可能是某种形式。它会强化或者鼓励某个人或者某个事物以更高的可能性产生同样的行为。当然，这里的行为不只是让某个人或事物去做某些事情，也可能是不去做某些事情，具体产生什么样的行为取决于使用什么样的“reinforcement”。

仅供非商业用途或交流学习使用

1.1.1 巴浦洛夫的狗

简单地说，强化学习的完整含义是利用这种“套路”完成学习。这个抽象的概念还有一个心理学上的解释，不禁让笔者想起了一个经典的心理学实验：巴浦洛夫的狗。在这个经典的实验中，每次实验者都对着狗摇铃铛，并给狗一点食物。久而久之，铃铛和食物的组合影响了狗的行为，此后每次对着狗摇铃铛，狗就会不由自主地流口水，并期待食物的到来，这样实验者就让狗“学会”了铃铛和食物的关系，这也可以算作强化学习的一个简单的例子。

了解了这个套路的一个具体例子，我们现在思考的是：能不能将这个套路的活动过程抽象成一个框架，使我们能更方便地将这个框架应用到更多的实例上呢？经过前辈们的苦心研究，最终我们得到了一个具有很高抽象度的强化学习框架。在“巴浦洛夫的狗”实验中，我们发现了下面几个关键要素。

- 狗：实验的主角。
- 实验者：负责操控和运转实验。
- 铃铛：给狗的一个刺激。
- 口水：狗对刺激的反应。
- 食物：给狗的奖励，也是改变狗行为的关键。

接下来，我们要给上面的每个要素赋予一个抽象的名字。

- 实验的主角：**Agent**。这个词不好翻译，一般翻译为**智能体**，但是这个翻译更像意译，和英文单词本身的含义有点距离。因此，本书将主要使用英文单词 Agent；在一些游戏问题中，为了更贴切行文环境，会以“玩家”来表示。
- 实验的操控者：**System Environment**，一般翻译为**系统环境**。
- 给 Agent 的刺激（铃铛）：**Observation**，一般翻译为**观察值**，但是在很多强化学习文献中，会将其等价为**状态**（**State**）。
- Agent 的反应（口水）：**Action**，一般翻译为**行动**。
- Agent 的奖励（食物）：**Reward**，一般翻译为**回报**或者**反馈**。

在经典的强化学习中，智能体要和环境完成一系列的交互。

（1）在每一个时刻，环境都将处于一种状态。

（2）智能体将设法得到环境当前状态的观测值。

（3）智能体根据观测值，结合自己历史的行为准则（一般称为**策略**，**Policy**）做出行动。

仅供非商业用途或交流学习使用

（4）这个行动会影响环境的状态，使环境发生一定的改变。Agent 将从改变后的环境中得到两部分信息：新的环境观测值和行为给出的回报。这个回报可以是正向的，也可以是负向的。这样 Agent 就可以根据新的观测值做出新的行动，这个过程如图 1-1 所示。

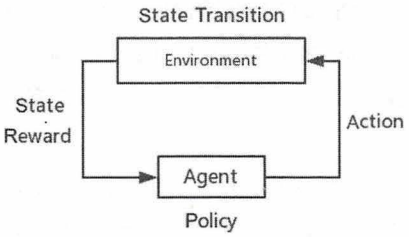


图 1-1　强化学习的过程表示

可以想象，在实验的早期，当实验者对着狗摇铃铛时，狗并不会有任何准备进食的反应；随着实验的进行，铃铛和食物这两个观测内容不断地刺激狗，使狗最终提高了准备进食这个行动的可能性。实际上，这样的交互模式在很多场景下都会出现。

1.1.2　俄罗斯方块

我们以一个经典的电子游戏俄罗斯方块为例，回顾强化学习的整个过程。常见的游戏界面如图 1-2 所示。

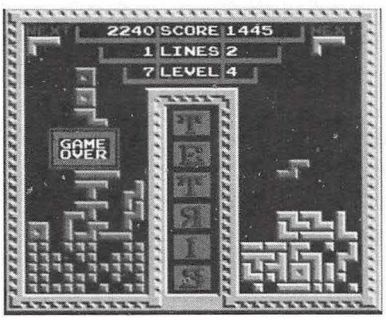


图 1-2　游戏俄罗斯方块的界面

以与巴浦洛夫的狗相同的角度看待俄罗斯方块游戏，我们发现此时的实验者就是游戏制作团队，而被实验的对象就是玩家。实验者通过构建一个俄罗斯方块游戏的环境，强化玩家一些行为的可能性。控制玩家行为的关键在于游戏的得分机制，由于我们明确了玩家的目标是尽可能地获得更高的分数，那么游戏团队对什么样的行为强化，玩家就倾向于产生什么样的行为。

仅供非商业用途或交流学习使用

从上面的描述中我们发现，如果游戏团队对消除方块的行动给予强化，那么玩家将更倾向于产生消除方块的行为；如果游戏团队对不消除方块的行动给予强化，玩家会更倾向于产生不消除方块的行为，当然这样会更快地结束游戏。如果游戏团队只对某种特定的消除方块行为（例如，用“竖条”一次性消除多行方块）给予较大的强化，那么玩家会尽可能地产生这种行为，其他消除方块技巧的使用频率会降低。实际上，俄罗斯方块这款游戏就是采用了这样的设计方案。

我们通过两个例子阐述了强化学习的核心思想：通过某种手段影响被实验者的行为。为了实现这个目标，实验者需要构建一个完整的实验环境，通过给予被实验者一定的观测和回报，让其产生实验者想要的结果。理解了这个思想，就算迈进了强化学习的大门。本书后面的内容就是剖析如何施展这些手段。

虽然本书从心理学的角度切入介绍了强化学习的概念，但毕竟是一本介绍人工智能技术的书籍，所以本书的“被实验者”全部是计算机模型或者仿真机器人等非有机生物。实际上，上面的实验框架也可以应用在人类或者动物身上，此时我们就转变成了研究心理学、社会学、管理学等人文学科的学者。因此从本质上看，文科和理科的界线并不是非常明确。如果你是一名对上述人文学科感兴趣的读者，不妨使用本节介绍的思想套用到这些学科，相信你会有意想不到的收获。

1.2 站在被实验者的角度看问题

看到这里，相信读者对强化学习有了一定的了解，前面我们提到的例子都是站在局外人的角度分析的，读者并没有参与到实验本身；而强化学习的定义是站在实验者的角度给出的，被实验者更像是一只被操纵的“木偶”。虽然这种控制他物的感觉十分奇妙，但是现实往往会将我们打回原形。实际上在日常生活中，我们往往会扮演“被实验者”这个角色。以俄罗斯方块为例，我们往往不是在扮演一个游戏的开发者，站在制高点控制玩家的行为，而是扮演一个玩家，思考如何才能玩好这个游戏。就算是游戏的开发者，也不能单纯地通过游戏影响别人的行为，他其实身处另外一个游戏，在这个游戏中，他需要通过制作优秀的游戏让更多人爱上自己的游戏，从而获得更多的收入和更大的影响力。从这个角度观察人类的行为，有时不免觉得悲哀。

让我们停止这样的思考，回到一个被实验者应有的视角。站在俄罗斯方块玩家的角度，我们的目标自然是尽可能地多得分。从游戏界面中，我们只能知道当前方块累积的效果和下一个到来的方块，而且所有的操作要在有限的时间内完成。除此之外，还有一些隐含的信息需要靠我们持续地观察才能得到，例如游戏中方块自然下落的速度。游戏的制作者可以从引擎中找到对应的参数，而作为玩家，我们只能通过画面变换的

仅供非商业用途或交流学习使用

速度进行判断。

这个差异可以由强化学习中的两个抽象实体表示。在 1.1.1 节的介绍中我们提到了两个概念：状态和观测值。其中状态表示 Agent 所处环境的所有信息，对俄罗斯方块这个游戏来说，等同于与游戏相关的所有信息；而观测值只是作为玩家能看到的一部分信息，两者实际上存在很大的差异。当然，对俄罗斯方块来说，由于游戏设计的区域相对有限，而且作为玩家我们可以知道下一个方块的信息，实际上信息量还算足够。

但是在另外一些问题中，状态和观测值存在很大的差异，这个差异会影响玩家的操作。以一些角色扮演游戏（Role-Playing Game，简称 RPG）的迷宫为例，从玩家的角度只能看到所处位置附近的信息，而这只占完整迷宫的一小部分，这时状态和观测的差异会对决策产生很大影响，由于每一时刻只能观测到部分信息，玩家需要记住其他区域的信息，才能更好地做出全局最优的决策。笔者以游戏《仙剑奇侠传 3》中的经典迷宫——蜀山锁妖塔第 4 层为例，展示两者的不同。迷宫的抽象版地图如图 1-3 所示。

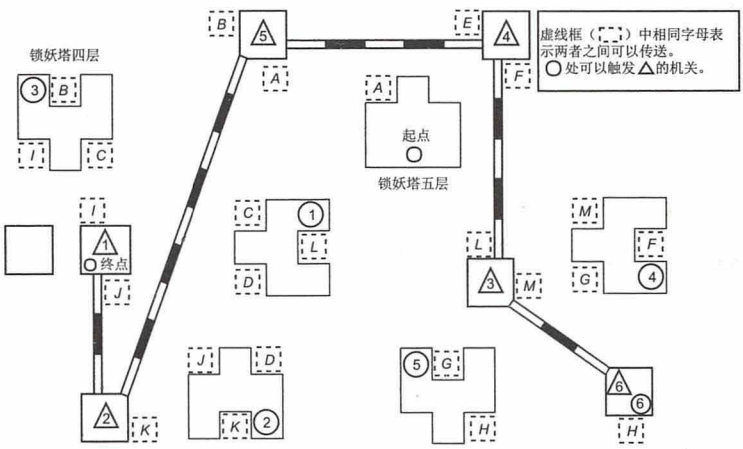


图 1-3　玩家制作的本层迷宫通关指南

地图中标示了几个主要的物体，其中以三角形 + 数字组合标示的物体为浮台，关卡开始时它在游戏空间下方，当玩家触发了圆圈 + 数字组合标示的机关，浮台就会升起。虚线框 + 字母组合标示的物体是传送台，相同字母的一对传送台可以将玩家传送到对应的位置。

想要通过这个迷宫，玩家必须按照顺序触动机关将 1~5 这几个浮台升起，并站在 6 号浮台上同它一起升起，然后从游戏空间上方通过进入下一关。要想把这些浮台升起，玩家需要在游戏空间下方穿梭于几个浮岛，触动每一个浮台的机关。如果读者多花一点时间分析这张图，就可以看出玩家需要按一定的顺序触发机关，如果触发顺序错误，将无法把所有的浮台升起。例如，如果玩家先把 5 号浮台升起，就无法到达图中左上角

仅供非商业用途或交流学习使用

的那个浮岛，也就无法触发 3 号浮台。

相信玩家可以感受到这个迷宫的复杂性，即使以这样抽象的形式表示出来，我们仍然需要经过一定的分析才能过关。在真实的游戏中，玩家根本不会看到这样简洁的迷宫地图，玩家的游戏界面如图 1-4 所示。

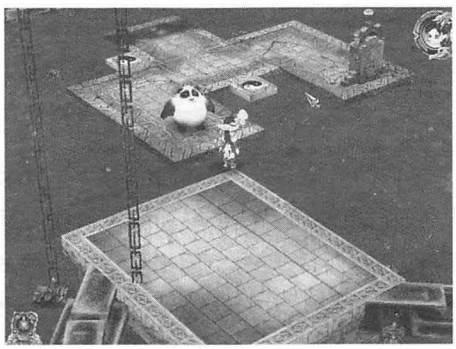


图 1-4 迷宫游戏截图

游戏的主界面下玩家只能看到一个浮岛的情况，而且传送台上并未标明它前往的目标。虽然在画面的左下角我们可以看到一个小地图，但是这个地图没有展现迷宫所有的内容，玩家想通过迷宫就变得非常困难。

通过上面的例子，相信读者已经了解了环境状态和观测的区别，也了解了很多问题的难度。在很多强化学习问题中，我们并不刻意区分这两个名词，而是将这两个名词混合使用。实际上，观测值也可以理解为 Agent 眼中环境的状态，希望读者可以注意这一点。

既然我们现在扮演了 Agent 的角色，我们的目标实际上发生了一定的变化，前面我们站在实验者的角度，目标是通过设计好的实验影响实验者的行为，而现在我们站在被实验者的角度，目标就变成了怎样又快又好地达到实验者的目标——改变行为，本书介绍的算法也是要达到这样的目标。

“又快又好地达到实验者的目标”这句话实在有些抽象，而且还是站在实验者的角度看待问题，我们需要一个更具体的目标。从前面的介绍中我们知道，实验者会通过一些手段强化 Agent 的某些行动，这个手段就是回报。如果 Agent 执行了实验者想要的行动，实验者就会给 Agent 更多的回报。反过来说，如果 Agent 配合实验者尽可能地将行为改变成实验者想要的效果，那么 Agent 就能获得最多的回报。所以站在 Agent 的角度，我们的目标是最大化所获得的回报，这个回报包含了 Agent 与环境交互的所有回报。这样，我们从观察内容和目标两个方面重新定义了被实验者眼中的强化学习。

仅供非商业用途或交流学习使用

1.3 强化学习效果的评估

1.2 节我们转换了强化学习的关注角度，并明确了学习的目标：最大化所获得的回报。但是这个目标还是有些抽象，我们需要把这个目标变得更容易量化。首先，我们介绍强化学习的两个显著特点，正是这两个特点使得它与其他学习方法存在不同。

1.3.1 不断试错

前面提到强化学习通过一些手段影响 Agent 的行动，这实际上是站在环境或实验者的角度来看的。如果站在 Agent 的角度，就是另外一幅景象：根据环境状态给出行动的 Agent 有时会收到较多回报，有时回报较少，回报可以以数值的形式标示，那么 Agent 还可能收到负的回报。究竟怎样才能获得最多的回报呢？Agent 自己并不知道，“实验者”也不会告诉 Agent。所以 Agent 需要根据回报的多少不断地调整自己的策略，从而尽可能多地获得回报。这个过程中 Agent 需要不断尝试，尝试应对状态的各种可能的行动，并收集对应的回报，只有收集到各种反馈信息，才能更好地完成学习任务。因此这是一个不断试错（Trial and Error）的过程，只有经过尝试、遇到失败，才能获得最终的成功。

1.3.2 看重长期回报

强化学习的任务通常需要长时间的交互，比如上面提到的俄罗斯方块游戏，玩家与游戏交互的周期可以是一局游戏。在这样的时间跨度下，眼前一步或两步操作获得的回报就变得没那么重要了。前面也提到，俄罗斯方块对同时消除不同层数方块的奖励不同，分 4 次消除一行方块得到的分数远远小于一次消除 4 行方块得到的分数。消除一行方块比较简单，只需要几个方块就可以做到；而消除 4 行方块就比较困难，需要前期方块的累积和准确地摆放一根竖棍。更难的方块摆放也使玩家更快地积累分数，从而更快赢得胜利，这和游戏设计者的初衷一致。因此，从更长远的角度看，玩家要学习如何一次消除多行，而不是追逐短期的得分。当然，追求长期分数需要多探索、多尝试，也可能遇到更多的失败（例如没等到竖棍出现），所以看重长期回报和不断试错存在一定的一致性。

正因为这两个特点，我们在评价强化学习算法的优劣上与其他算法不同。除了一些常见的衡量指标（算法的效果、计算时间、稳定性和泛化性等），我们还要重点考虑一个指标：学习时间。由于学习和尝试相关，所以这个指标一般也看作尝试和探索的

仅供非商业用途或交流学习使用

次数。如果一个算法需要尝试的次数比较多，我们一般认为算法要花费的时间比较长；如果一个算法需要尝试的次数比较少，那么相对来说花费的时间比较短。

站在机器学习的角度，我们可以认为尝试的样本本身会影响学习的时间，例如样本的代表性、重合度等。对强化学习来说，由于学习本身的特点，我们需要考虑训练样本的使用率（Sample Efficiency）。不同算法对样本的重复使用次数不同，有的算法对于尝试的样本只能使用一次，而有的算法可以反复使用同样的样本。这些算法的特点我们会在后面的章节介绍。

训练样本的使用率会直接影响学习时间。前面提到 Agent 的学习样本要通过自身与环境的交互得到，而这个过程是要花费时间的。需要的样本量少，学习时间就可以缩短；反之学习时间会比较长。对计算机模拟的学习问题来说，样本量并不算个大问题，因为计算机可以在短时间内快速模拟出大量的样本；但是对于在真实场景进行训练的问题来说，产生样本意味着要在真实世界的时间尺度下进行交互，花费的时间会很长。为了一点效果的提升花费大量的时间，对我们来说有点得不偿失。因此，很多研究人员都在思考如何提高真实世界学习的速度，这就涉及提高样本利用率、迁移学习等内容。

这样我们就了解了强化学习关注的两个目标：学习效果和学习时间。学习时间也成了算法十分看重的一个目标。

1.4 强化学习与监督学习

有些读者会有疑问：强化学习和其他常见的方法有什么相同与不同呢？有些读者应该知道监督学习，有些读者可能不太了解，监督学习是一种很经典的机器学习方法，它的目标是训练一个模型，使模型能够根据确定的输入得到对应的输出。为了实现这个目标，我们需要准备好一定数量的训练样本，这些样本包含了一对对的输入和输出数据，我们使用输入和输出数据计算得到模型的参数，从而完成模型的学习。

1.4.1 强化学习与监督学习的本质

从学习的目标来看，两个学习方法十分相近。监督学习希望模型根据指定的输入得到对应的输出，强化学习希望 Agent（也可以想象成模型）根据指定的状态得到使回报最大化的行动。看上去二者都是完成从一个事物到另一个事物的映射，但是只要做进一步的分析，我们就可以梳理出它们的不同。

仅供非商业用途或交流学习使用

1. 二者在目标的明确性上有所不同

监督学习的目标更明确，输入对应的是确定的输出，而且理论上一个输入只对应一个输出；而强化学习的目标没有这么明确，使当前状态获得最大回报的行动可能有很多。

以俄罗斯方块为例。如果采用监督学习进行训练，那么模型以每一帧游戏画面或游戏状态为输入，对应的输出是确定的，要么移动方块，要么翻转方块。这样的限定实际上有些死板，某个操作序列能够达到想要的目标（例如得更多的分），但是得更多的分数的方法不止一种。强化学习直接设定某个目标，并根据这个目标设定反馈，这样的限定相对宽松，定义问题的难度也更低。

2. 从时间维度上看，二者输出的意义不同

监督学习主要看重输入和输出的匹配程度，如果输入和输出匹配，学习的效果就是好的，即使存在序列到序列的映射，我们也希望每一个时刻的输出都能和输入对应上；而对强化学习而言，我们的目标是让回报最大化，但是在与环境交互的过程中，并不是每一个行动都会获得回报（本书中也会涉及一些这样的任务）。当我们完成了一次完整的交互后，会得到一个行动序列，这个行动序列中哪些行动为回报产生了正向的贡献，哪些产生了负向的贡献，有时很难界定。

以棋类游戏为例，游戏的目标是战胜对方（获得胜利），在得到最终结果之前，Agent 可能不会得到任何回报。单一的行动是优是劣无从判断，我们只能把所有的行动考虑成一个整体，给整体一个回报。为了最终的回报，游戏中的某些行动可能会走一些看似不好的招法（例如被对方吃掉棋子），这是为达成最终目标做出的牺牲。从这个角度来看，强化学习和监督学习存在一定的不同，这个不同也将在后面的章节展开介绍。

总的来说，强化学习相比监督学习有两个优点。

（1）定义模型需要的约束更少，影响行动的反馈虽然不及监督学习直接，却降低了定义问题的难度。

（2）更看重行动序列带来的整体回报，而不是单步行动的一致性。

1.4.2 模仿学习

实际上有一种学习方法的目标和强化学习一致，都是最大化长期回报，但是学习方法和监督学习类似，收集大量的单步决策样本，并让模型学习这些单步决策的逻辑。

仅供非商业用途或交流学习使用

这种方法被称为“模仿学习”（Imitation Learning），也被称为“行为克隆”（Behavior Cloning），它是模仿学习的一种。它的执行流程为：

（1）寻找一些“专家”（Expert）代替 Agent 与环境交互，得到一系列的交互序列。

（2）假设这些交互序列为对应状态下的“标准答案”，我们就可以使用监督学习的方法让模型学习这些数据，从而完成将状态和专家行动对应的工作。

这种方法在一些问题上可以获得比较好的效果，但也存在如下问题。

（1）如何收集满足目标的样本？我们必须在某一个领域存在一个专家，所有的样本通过专家和环境的交互产生。

（2）如何收集大量的样本？如果样本数量不够多，我们很难学习出好的策略模型。

（3）如何确保学习的模型拥有足够的泛化性？模仿学习常常会遇到“分布漂移”（Distributional Shift）的问题，这与模型的泛化性有一定的关系。我们的样本通常无法覆盖问题的所有情况，在实际使用过程中必然会出现一些训练样本中没有出现的观测值，导致泛化能力不强的 Agent 出现重大失误。为了解决这个问题，我们需要从模型和训练样本两方面入手。

以上三个问题都不容易解决，因此模仿学习的难度并不小，这才使得大家把目光集中在强化学习上，希望它能够解决模仿学习无法解决的问题。

1.5 强化学习的实验环境

前面我们介绍了强化学习的概念，可以看出强化学习相比监督学习有更多的优势，可以应用到更广的范围中。现在已知的强化学习问题有很多，例如，解决一些长期决策问题、博弈问题，使机器人模拟真实人类完成一些特定动作等。

由于强化学习需要不断地在环境中尝试，因此一些问题的学习效率成了挑战。例如，仿真机器人模拟人类行为这样的学习，让仿真机器人一遍又一遍尝试人类的动作是比较慢的，可能要花很长时间才能让机器人学到期望的行动，但如果将学习的内容转移到计算机上，通过模拟的方式进行学习，速度将快许多。因此，目前大部分的强化学习是在计算机上进行的。

为了让更多人快速地应用强化学习，也为了大家能够针对同样的问题讨论交流，业界开放了一批可以公开访问搭建的实验环境，这里有我们常见的计算机游戏环境，也有机器人仿真的环境，本节介绍一些广为人知的环境。

仅供非商业用途或交流学习使用

1.5.1 Arcade Learning Environment

Arcade Learning Environment（简称 ALE）环境封装了很多基于 Atari 2600 游戏机且曾经知名的游戏，例如乒乓球游戏（Pong），游戏界面如图 1-5 所示。

游戏的双方控制一个板子，通过反弹的方式将乒乓球弹到对方那边，如果哪一方没有接到球，对方就可以得 1 分。在一局对弈中，如果一方率先得到 21 分，这一方就获得了比赛的胜利。从游戏界面可以看出，这个游戏只需要玩家操纵“上”和“下”两个按键，游戏的规则和操作十分简单，最终的决策结果也是离散的形式。

当年的游戏是在一个很小的屏幕上展示的，屏幕的宽为 160 像素，高为 210 像素，每个像素有 128 个颜色。游戏机共有 18 种操作方式，所有的操作都是通过图 1-6 所示的简单的游戏杆和按钮完成的。

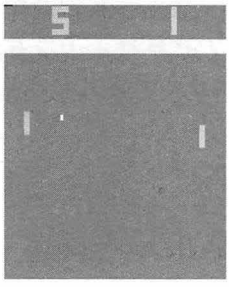


图 1-5 Pong 游戏的界面



图 1-6 Atari 2600 游戏机

ALE 是基于另一个开源的 Atari 2600 模拟器构建完成的。这个模拟器实现了简单的操作接口，我们可以给模拟器发送指定的被离散化后的操作，模拟器会将模拟得到的最近的屏幕信息返回。在实时状态下，模拟器可以以每秒 60 帧的速度生成游戏画面，还可以在更高的帧率生成画面 [1]。ALE 还提供保存当前游戏模拟器状态的功能，这个功能就像存档一样，把当时游戏的状态保存下来，这样即使游戏继续进行，也可以回到当时的场景重新进行游戏。这个功能可以帮助研究环境已知（Model-based）方法的研究人员，让他们在长期规划（Long Time Planning）方面所有突破。

1.5.2 Box2D

前面提到的游戏环境更像在一个虚幻的世界里构建了一些物理法则，而对接下来要介绍的 Box2D 环境来说，它更多地遵从了真实世界的物理规则。Box2D 是一个模拟刚体运动的环境库，使用它，我们可以在游戏中创造出类似真实世界的物体运动效果，

仅供非商业用途或交流学习使用

如图 1-7 所示。所谓刚体，是指这个物体在经历任何运动后都不会发生形变（官方的定义为：物体上任意两点的位置永远保持固定）。这个环境中还包含摩擦力、重力、碰撞等物体运动的效果。

Gym 中也有一些基于 Box2D 的游戏，例如月球登陆者（Lunar Lander），如图 1-8 所示。这个游戏通过控制登陆器的平衡使登陆器稳定地降落在指定的位置。这其中涉及很多物理环境的构建。

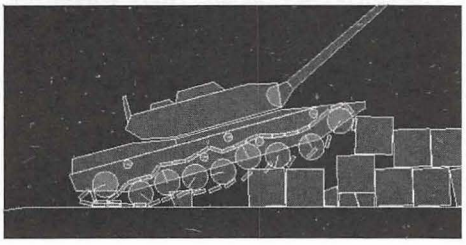


图 1-7 使用 Box2D 实现的模拟场景

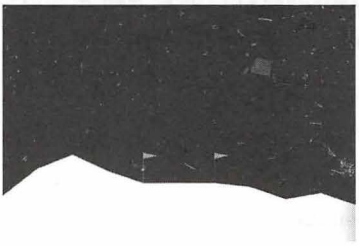


图 1-8 Gym 中展示的 Box2D 游戏

1.5.3 MuJoCo

MuJoCo[2] 是一个 3D 环境下的物理仿真引擎，它很好地模拟了三维环境下的一些物理性质。我们可以使用这个引擎设计一些仿真环境，训练模型实现一些近似现实生活中的任务。它和 Box2D 的功能类似。图 1-9 展示了利用 MuJoCo 实现的物理环境，我们可以通过操纵图 1-10 所示的人体结构的关节和骨骼，实现一些仿人的任务。如果模型在仿真环境上得到了验证，就可以应用到现实世界中的机器人上，让机器人进行进一步适应性训练也会容易些。

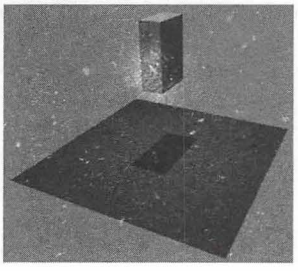


图 1-9 MuJoCo 的简单环境实现

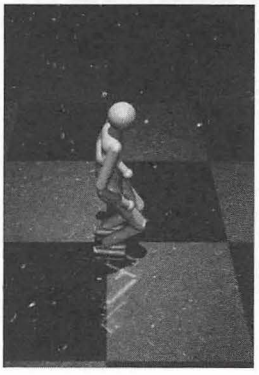


图 1-10 3D 场景下的仿人类行动实验

仅供非商业用途或交流学习使用

1.5.4 Gym

Gym 是一个集成了众多强化学习实验环境的平台，拥有这个平台，我们可以很方便地搭建起强化学习需要的环境，从而集中精力完成策略部分的工作。在 Gym 诞生之前，已经有一些优秀的强化学习平台存在，但这些平台相对比较分散。Gym 将这些平台的模拟环境做了整合，使环境配置变得简单。

Gym 的设计目标是为研究人员提供一个易于操作的环境，例如，Atari 里的经典游戏 Pong，仅需运行下面的代码：

```
import gym
env = gym.make('Pong-v0')
env.reset()
env.render()
```

就可以搭建游戏环境。至于模型的策略该如何执行，则没有包含在其中，不过他们在另外一个项目中提供了对基本策略的实现，这个项目叫作 Baselines。

同时，为了方便强化学习研究人员的沟通交流，Gym 的官方网站还提供了一个“打擂”平台，通过官方提供的一些开发库，每个人都可以记录自己算法的一些细节（例如，每一轮学习花费的时间、为了学习进行的交互数量）。同时，每个人都可以将自己实现的代码和结果上传到对应的强化学习问题页面上，这样既可以互相学习，又促进了对问题的深入研究，如图 1-11 所示。

CartPole-v1 Evaluations

ALGORITHM	EPISODES BEFORE SOLVE	SUBMITTED
ruippeixotog's algorithm writeup	1052.0	2017-09-09 22:41:43.170805
tomchang25's algorithm writeup	N/A	2017-09-04 02:52:28.480129
planktonfun's algorithm	0.0	2017-09-01 14:37:45.751573
goofool's algorithm	0.0	2017-08-31 10:07:03.623937
haron1100's algorithm	1.0	2017-09-11 01:59:22.365577
haron1100's algorithm	6.0	2017-09-11 14:30:42.406691
haron1100's algorithm	6.0	2017-09-11 14:27:11.856827

图 1-11　Gym 的效果展示平台

从页面上可以看出，Gym 不仅关注模型的最终效果（即最终得到的回报总值），还

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

关注学习模型所使用的样本数量，而样本数量和学习时间相对应，这也和我们提到的强化学习的目标相呼应。关于 Gym 和 Baselines 的内容将在本书的第 5 章详细介绍。

1.6 本书的主要内容

强化学习在机器学习中的难度不低，它需要很多方面的知识辅助，同时自身也已经形成了一个庞大的体系。本书并不能帮助读者完全掌握强化学习的所有知识点，所能做的只是展示其中部分基础内容。本书不是一本科普读物，想要阅读本书需要具备一定的基础知识，如微积分、线性代数等。部分章节也会梳理这些基础知识，以确保读者掌握这些知识的核心思想。本书各章节的核心内容如下。

第一部分主要介绍与强化学习有关的基础知识，例如数学基础、对应的程序开发基础、强化学习的基本计算方法等。

第 1 章已经介绍了强化学习的基本概念，相信读者对强化学习的目标、特点已经有了一定的了解。

第 2 章介绍相关的数学知识，如线性代数、概率论、重要性采样、信息论等，帮助读者快速回顾即将用到的数学知识。

第 3 章介绍强化学习中会用到的优化知识，主要介绍常见的梯度下降法（Gradient Descent）、共轭梯度法和自然梯度法，这三种方法将出现在后面的算法中。

第 4 章介绍书中代码使用的计算框架 TensorFlow。TensorFlow 是一款使用十分广泛的框架，很多强化学习的算法选择使用它进行实现，因此我们有必要学习它。本章将简单介绍它的使用方法和一些基本原理，熟悉 TensorFlow 的读者可以跳过本章。

第 5 章介绍本书使用的另一个框架 Gym 及在此框架上实现的算法集合 Baselines。Gym 集成了大量的强化学习仿真环境，Baselines 则基于 TensorFlow 和 Gym 实现了一些经典的算法。本章将简单介绍这两个框架的基础知识。

第 6 章介绍强化学习的基础知识。例如马尔可夫决策过程（Markov Decision Process），以及在简单问题上的两种经典动态规划求解法：策略迭代法和价值迭代法。这些方法是强化学习算法的基石，绝大多数强化学习方法都是根据这些知识演变来的。

第二部分介绍以最优值函数为思想的一系列算法，其中的代表算法为 Deep Q Network 和 Rainbow。

第 7 章介绍蒙特卡罗（Monte-Carlo）和时序差分（Temporal-Difference）两种求解 Model-free 问题的方法，并介绍 Deep Q Network 算法的细节。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

第 8 章介绍 Deep Q-Learning 的一些改进算法，如 Priority Replay Buffer、Duel Network 等，并介绍改进的集成算法 Rainbow。

第三部分介绍以策略梯度（Policy Gradient）为思想的一系列算法，其中的代表算法为 Actor-Critic。

第 9 章介绍策略梯度法和 Actor-Critic 算法的原理，同时介绍 A2C 算法的实现细节。

第 10 章介绍使策略单调提升的算法，其中的代表为 Trust Region Policy Optimization（TRPO）算法。

第 11 章介绍高样本使用率的策略梯度算法，其中的代表算法为 ACER 算法和确定策略梯度法（Deterministic Policy Gradient）。

第四部分介绍强化学习其他方面的内容。

第 12 章介绍回报稀疏情况下的一些求解方法，其中包括基于层次的强化学习和基于课程学习（Curriculum Learning）思想的方法。

第 13 章介绍模型已知的一些算法，如基于蒙特卡罗树搜索（Monte Carlo Tree Search，MCTS）的强化学习算法和 iLQR 算法的原理。

第五部分介绍反向强化学习的基础知识。

第 14 章介绍反向强化学习的基础，以及基本的求解方法。

第 15 章介绍最大熵反向强化学习（Max Entropy Inverse Reinforcement Learning）和生成对抗模仿学习（Generative Adversarial Imitation Learning）算法的内容。

虽然本书介绍了很多强化学习的内容，但实际上这只是强化学习中的一小部分，不过这些内容可以作为读者入门强化学习的基础，希望在下面的学习中读者能够有所收获，并以此为基础了解更多有关强化学习的内容。

1.7 参考资料

[1] Naddaf Y, Naddaf Y, Veness J, et al. The arcade learning environment: an evaluation platform for general agents[J]. Journal of Artificial Intelligence Research, 2012, 47(1):253-279.

[2] Todorov E, Erez T, Tassa Y. MuJoCo: A physics engine for model-based control[C]// Ieee/rsj International Conference on Intelligent Robots and Systems. IEEE, 2012:5026-5033.

[3] Brockman G, Cheung V, Pettersson L, et al. OpenAI Gym[J]. 2016.

[4] Box2D v2.3.0 User Manual http://box2d.org/manual.pdf

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2 数学与机器学习基础

在正式开始强化学习之旅前，我们还是要对数学和机器学习的基础知识做一些介绍。不同于第 1 章生动的文字描述，从这一刻开始书中将出现令人讨厌的数学公式。物理学界泰斗霍金曾说过："每多写一个公式就会吓跑一半读者"，但愿这句话不要成为现实。下面的一些内容总体上比较基础，有这方面基础的读者可以跳过本章，想要温习基础知识的读者可以仔细阅读。总而言之，看完本章，我们在机器学习基础方面将取得很多共识，阅读本书后面的章节时不会被突然出现的名词搞晕。

2.1 线性代数基础

线性代数是一门与机器学习紧密相关的数学课程，其中的很多定理、性质和方法论在机器学习中起到了关键性的作用。本节旨在简单回顾线性代数中的一些基本内容，为后面的内容做铺垫。实际上，这些内容都被涵盖在大学的线性代数课本中，只是大学课本没有专门围绕着机器学习内容进行讲述，下面的内容将围绕着线性变换的核心概念展开。

在线性代数中，大家都接触过"**矩阵**"这个概念，对于计算机专业的读者来说，矩阵 $\boldsymbol{A}_{m\times n}$ 可以看作一个二维数组 $\boldsymbol{A}$[m][n]，其中第一维表示行，第二维表示列。矩阵也拥有自己的运算体系，相同维度的矩阵可以相加，矩阵可以和标量相乘，在维度匹配的情况下，矩阵与矩阵之间也可以相乘。矩阵运算满足加法交换律、乘法结合律，不满足乘法交换律。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

矩阵运算中经常会遇到一些特殊操作，例如**转置**。矩阵 $\boldsymbol{A}_{m\times n}$ 经过转置后就变成了矩阵 $\boldsymbol{A}^{\mathrm{T}}_{n\times m}$，相当于矩阵沿着主对角线进行翻转得到了新的矩阵。如果有一个矩阵，它的转置和它本身相同，那么这个矩阵就被称为**对称矩阵**。这类矩阵将成为本书中的一大主角。

矩阵的另一类运算是除法运算。并不是每一个矩阵都可以被除，在线性代数中，如果一个矩阵可以被除，那么就称这个矩阵是**可逆的**。关于矩阵可逆的判断条件还有很多，这里就不一一介绍了。

线性代数中最基本也是最经典的一类运算就是**矩阵和向量的乘法**。如果有矩阵 $\boldsymbol{A}_{m\times n}$ 和向量 $\boldsymbol{x}_{n\times 1}$，那么这两者是可以相乘的，有

$$\boldsymbol{A}_{m\times n}\cdot\boldsymbol{x}_{n\times 1}=\boldsymbol{b}_{m\times 1}$$

这个公式实际上和机器学习中的一些线性模型相似。如果把矩阵 $\boldsymbol{A}$ 想象成一个运算符，那么这个运算符相当于对一个 n 维的向量 $\boldsymbol{x}$ 做运算，得到一个 m 维的向量 $\boldsymbol{b}$。如果考虑 $\boldsymbol{x}$ 存在于一个 n 维的实数向量空间，$\boldsymbol{b}$ 存在于一个 m 维的实数向量空间，那么 $\boldsymbol{A}$ 的作用相当于一个映射，将不同维度的向量关联起来，而且它们之间是线性计算的关系。因此，这种计算也可以被认为是**线性变换**。

线性变换可以从两个角度理解：首先从常规的运算方法来看，$\boldsymbol{b}$ 中的每一个元素都是由 $\boldsymbol{A}$ 中的一个行向量和 $\boldsymbol{x}$ 相乘得到的，它们的关系如图 2-1 所示。

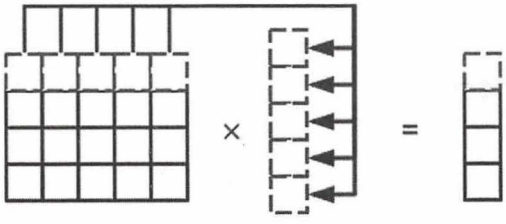


图 2-1　线性变换的一种运算形式

在这种运算形式中，$\boldsymbol{A}$ 的每一行可以想象成 $\boldsymbol{x}$ 中每一个元素的权重，它们计算的结果汇总成一个数，相当于对 $\boldsymbol{x}$ 的元素做加权求和，得到的结果代表了对 $\boldsymbol{x}$ 数据的汇总。所以计算中将 n 个数据做了 m 个汇总。如果再深入一点分析，这些汇总有什么含义呢？可以看出，这里的计算是两个向量的内积运算，内积运算具有自己的运算语意，假设有两个向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，那么它们的内积公式可以写作

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}||\boldsymbol{b}|\cos\theta$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

其中 θ 表示 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 在向量空间中的夹角。如果两个向量的长度（norm）为 1，那么两个向量的内积表示了两个向量的某种相关度。如果两个向量共线且同向，那么内积的结果为 1；如果两个向量相互正交，那么内积为 0；如果两个向量方向相反，那么内积为 -1。

所以从上面的分析中可以看出，线性变换相当于对 $\boldsymbol{x}$ 完成了 m 次内积运算，每一次内积运算的内容是在考察 $\boldsymbol{x}$ 与 $\boldsymbol{A}$ 的行向量的相关程度，所以 $\boldsymbol{b}$ 就是 $\boldsymbol{A}$ 与 $\boldsymbol{x}$ 的相关程度的汇总。因此作为一个运算符，$\boldsymbol{A}$ 的内容就显得十分关键了——它直接决定了“心目中”理想的向量：像 $\boldsymbol{A}$ 则结果会比较大，不像结果就会比较小。

另一种理解方式并不是运算时常用的方式，但却是理解线性代数很重要的一种方式。在这种方式中，运算过程将被重新组合。

（1）$\boldsymbol{A}$ 中的每一个列向量和 $\boldsymbol{x}$ 中的每一个元素依次相乘。

（2）相乘后的向量再加和在一起，得到结果 $\boldsymbol{b}$。

这种计算过程如图 2-2 所示。

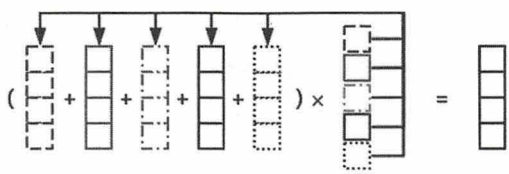


图 2-2 线性变换的另一种解释

在讨论向量的时候，我们在讨论什么？讨论向量在向量空间内的运算。向量空间有以下几种经典的运算方法：

（1）向量相加。

（2）向量与标量数字相乘。

（3）向量间求内积。

前面的第一种解释方法主要使用了第 3 种运算方法，后面的这种解释方法则主要使用了前面两种运算方法。

在这种解释下，$\boldsymbol{x}$ 的每一个元素变成了 $\boldsymbol{A}$ 列向量的权重，实际上这个线性变换变成了向量级别的加权平均。这种解释似乎更为直观。$\boldsymbol{A}$ 的每一个列向量表示了结果 $\boldsymbol{b}$ 所在空间的一种向量表达，那么 $\boldsymbol{x}$ 的作用就是平衡这些向量表达并将这些表达揉合成一个新的向量。

从这一种线性变换的角度来分析，我们就要引出另外一个概念——**线性独立**。从图 2-2 中读者已经了解了向量的加权平均，那么有没有可能经过加权求和得到其中的某一

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

个向量呢？当然可能。可以让 $\boldsymbol{x}$ 中与这个向量相对应的元素为 1，其他的元素为 0，那么结果就是这个向量。但是如果这里要求这个对应的元素为 0，只能通过其他向量的线性组合（也就是加权求和）来得到这个向量呢？这就不一定了，有些情况下这件事情是可以做到的，有些情况下则无法做到。以如下矩阵为例：

```
a = [
  [1 0 0 0]
  [0 1 0 0]
  [0 0 1 0]
  [0 0 0 1]
]
```

在这个矩阵里，任意一个列向量都不能被其他列向量经过加权求和得到。这样线性独立的概念就比较明确了，如果说一组向量是线性独立的，那么其中的任意一个向量都不能被其他向量通过加权相加得到。

这个性质有什么用处呢？实际上在对线性代数的介绍中很多概念已经被省略，例如矩阵的秩。本节不去考虑那些概念，只是用刚才的运算做进一步的分析。对于 $\boldsymbol{Ax} = \boldsymbol{b}$ 这样的运算，如果 $\boldsymbol{A}$ 中的列向量组不是线性独立的，那么为了求出结果 $\boldsymbol{b}$，我们可以使用更小维度的 $\boldsymbol{A}$ 和 $\boldsymbol{x}$。比如说现在有如下的运算：

```
import numpy as np
A = np.array([
  [1, 0, 0]
  [0, 1, 0]
  [1, 1, 0]
])
b = np.array([1, 1, 1])
np.dot(A.T, b)
# array([2, 2, 0])
```

如果想要得到同样的结果，我们可以缩小 $\boldsymbol{A}$ 的维度：

```
A = np.array([
  [1, 0, 0]
  [0, 1, 0]
])
b = np.array([2, 2, 0])
np.dot(A.T, b)
# array([2, 2, 0])
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

这说明完成同一个运算时，矩阵 $\boldsymbol{A}$ 可以变得更小。上面第一段代码中，$\boldsymbol{A}$ 是一个 3×3 的矩阵，里面包含着 3 个三维的列向量，而第二段代码中，$\boldsymbol{A}$ 是一个 2×3 的矩阵，里面包含着 3 个二维的列向量，这说明上面一段代码中的矩阵 $\boldsymbol{A}$ 看似处于三维空间中，但实际上它的列向量只存在于其中的一个二维子空间中。

线性代数中有一种特殊的线性无关向量组，这个向量组就是**基**。基在英文中称为“basis”，这里也可以理解成“基础”的含义。最常见的基的形式就是欧式坐标轴体系中，那些坐标轴方向上的向量组成的向量组。基具有以下特点：

- 它们的组成部分——这些向量是线性无关的。
- 通过这些向量的线性组合，可以表示所在空间内的任意一个向量。以上面提到的二维空间为例，对于任意一个实数向量，都可以利用这组基通过线性变换表示出来：$(x, y) = x \cdot (1, 0) + y \cdot (0, 1)$。

除了线性相关与无关这样的关系，线性代数中还有一种常见的关系，被称为**正交**。如果一个矩阵和一个向量相乘的结果为 0 向量，而矩阵和向量都不全为 0，那么从线性变换的角度来看，可能有两种情况：

- 向量和矩阵的每一个行向量点乘为 0。
- 矩阵列向量经过被乘向量的线性组合后，相互抵消。

2.2 对称矩阵的性质

对称矩阵是线性代数中十分重要的一种矩阵类型，因此这里将它单独列出作为一节进行分析。

2.2.1 特征值与特征向量

前面提到了线性变换的运算，那么我们就要深入分析线性变换中的一类特殊的变换。让我们再回到上面的公式

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

此时这个公式对其中的两个向量还没有太多的约束，如果我们对公式增加如下两个约束。

（1）$\boldsymbol{x}$ 和 $\boldsymbol{b}$ 的维度相同，同为 n 维，那么 $\boldsymbol{A}$ 就是一个 $n \times n$ 的方阵，是 n 维空间内的一个算子。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

（2）$\boldsymbol{x}$ 和 $\boldsymbol{b}$ 共线，也就是说 $\boldsymbol{b}=\lambda \boldsymbol{x}$。

这里再将 $\boldsymbol{x}$ 换为这个公式里常用的符号——$\boldsymbol{e}$，那么上面的公式就可以变为

$$\boldsymbol{A}\boldsymbol{e}=\lambda \boldsymbol{e}$$

此时我们发现，经过 $\boldsymbol{A}$ 的线性变换后，向量 $\boldsymbol{e}$ 仅仅做了尺度的缩放，而没有做其他的变换，这时我们认为矩阵 $\boldsymbol{A}$ 和向量 $\boldsymbol{e}$ 存在某种特殊的关系，于是人们将向量 $\boldsymbol{e}$ 称为**特征向量**，尺度缩放的系数 λ 被称为**特征值**。

2.2.2 对称矩阵的特征值和特征向量

对称矩阵的特征值和特征向量都有自己的特点。首先，由**实数组成的对称矩阵的特征值全部为实数**，即特征值的共轭数等于自身，以下就来证明。

对于某个实数对称矩阵 $\boldsymbol{A}$，它的共轭矩阵 $\tilde{\boldsymbol{A}}=\boldsymbol{A}$，对于某个特征值 λ 和某个非 0 的特征向量 $\boldsymbol{e}$，我们有

$$\begin{aligned}\tilde{\boldsymbol{e}}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}&=\tilde{\boldsymbol{e}}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{e}\\&=(\boldsymbol{A}\tilde{\boldsymbol{e}})^{\mathrm{T}}\boldsymbol{e}\\&=(\tilde{\boldsymbol{A}}\tilde{\boldsymbol{e}})^{\mathrm{T}}\boldsymbol{e}\end{aligned}$$

所以等式两边有

$$\begin{aligned}\lambda\tilde{\boldsymbol{e}}^{\mathrm{T}}\tilde{\boldsymbol{e}}&=\tilde{\lambda}\tilde{\boldsymbol{e}}^{\mathrm{T}}\boldsymbol{e}\\(\lambda-\tilde{\lambda})\tilde{\boldsymbol{e}}^{\mathrm{T}}\boldsymbol{e}&=0\end{aligned}$$

因为 $\tilde{\boldsymbol{e}}^{\mathrm{T}}\boldsymbol{e}$ 不为 0，所以特征值和它的共轭相等，所以对称矩阵的特征值全为实数。

其次，**对称矩阵的特征向量相互正交**，证明如下。

对于对称矩阵中的两对不同的特征向量、特征值对 $\boldsymbol{e}_1,\lambda_1$ 和 $\boldsymbol{e}_2,\lambda_2$，可以推导出

$$\begin{aligned}\lambda_1\boldsymbol{e}_1\cdot\boldsymbol{e}_2&=(\lambda_1\boldsymbol{e}_1)^{\mathrm{T}}\boldsymbol{e}_2\\&=(\boldsymbol{A}\boldsymbol{e}_1)^{\mathrm{T}}\boldsymbol{e}_2\\&=\boldsymbol{e}_1^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{e}_2\\&=\boldsymbol{e}_1^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_2\\&=\boldsymbol{e}_1^{\mathrm{T}}(\lambda_2\boldsymbol{e}_2)\end{aligned}$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

所以有 $\lambda_1 \boldsymbol{e}_1 \boldsymbol{e}_2 = \lambda_2 \boldsymbol{e}_1 \boldsymbol{e}_2$。

因为 $\lambda_1 \neq \lambda_2$，所以 $\boldsymbol{e}_1 \boldsymbol{e}_2 = 0$。同理，我们可以推导出所有的特征向量对之间都正交，从而完成证明。

如果将对称矩阵所有特征向量的长度限制为 1（也就是进行标准化），那么这些特征向量组成的矩阵就成了一个标准正交矩阵。这个标准正交矩阵也拥有一个性质，那就是它的逆矩阵等于它的转置矩阵，也就是它自身，由此可得

$$\boldsymbol{E}\boldsymbol{E}^{\mathrm{T}} = \boldsymbol{I}$$

$$\boldsymbol{E}^{\mathrm{T}} = \boldsymbol{E}^{-1} = \boldsymbol{E}$$

2.2.3 对称矩阵的对角化

讲完了上面的两个性质，下面介绍对称矩阵的另一个性质。前面提到了线性变换操作，如果这一次执行线性变换的是一个对称矩阵，那么会发生什么情况呢？

首先，根据特征值与特征向量的定义，可知

$$\boldsymbol{A}\boldsymbol{e} = \lambda \boldsymbol{e}$$

将所有的特征值和特征向量融合到一起，并用 $\boldsymbol{\Lambda}$ 表示对角线为特征值，其余位置为 0 的对角阵，就有

$$\boldsymbol{A}\boldsymbol{E} = \boldsymbol{E}\boldsymbol{\Lambda}$$

2.2.2 节提到了对阵矩阵的特征向量组成了标准正交矩阵，于是有

$$\boldsymbol{A} = \boldsymbol{E}\boldsymbol{\Lambda}\boldsymbol{E}^{-1} = \boldsymbol{E}\boldsymbol{\Lambda}\boldsymbol{E}^{\mathrm{T}}$$

这个过程被称为对称矩阵的对角化，那么这个过程有什么作用呢？如果要求矩阵 $\boldsymbol{A}$ 的 2 次幂，对角化的结构可以帮助简化这个过程：

$$\boldsymbol{A}\boldsymbol{A} = (\boldsymbol{E}\boldsymbol{\Lambda}\boldsymbol{E}^{\mathrm{T}})(\boldsymbol{E}\boldsymbol{\Lambda}\boldsymbol{E}^{\mathrm{T}}) = \boldsymbol{E}\boldsymbol{\Lambda}^2\boldsymbol{E}^{\mathrm{T}} = \boldsymbol{E}\boldsymbol{\Lambda}^2\boldsymbol{E}$$

于是对于任意次阶乘，都有

$$\boldsymbol{A}^n = \boldsymbol{E}\boldsymbol{\Lambda}^n\boldsymbol{E}$$

由于阶乘只发生在对角阵上，于是计算变得简单了许多。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2.3 概率论

概率论也是机器学习中十分重要的一部分知识，可以说，当今机器学习的理论基础就是统计和概率论。如果要展开这部分的知识，恐怕一本书也讲不完，这里同样介绍最简单的概念。

2.3.1 概率与分布

阿甘的妈妈曾经说过："Life is like a box of chocolates, you never know what you're going to get."（电影《阿甘正传》台词）。生活中充满了不确定性，这些随机给我们带来了快乐，也带来了烦恼。概率论中的很大一部分工作就是描述这些不确定性。最先引出来的概念是**随机事件**，一个随机事件就是一件充满了不确定性的事情，比如说明天下雨这件事。明天可能下雨，也可能不下雨，那么到底下不下雨呢？这就要靠概率来描述了。概率一般用 $P(X)$ 表示，括号里的内容 X 就是计算概率的实体。为了度量的方便，概率值的范围被限定在 $[0, 1]$。概率值为 0 表示事件完全不会发生，概率值为 1 表示事件一定会发生。那么随机事件的概率值就可以写作

$$P(\text{明天下雨}) = 0.4$$
$$P(\text{明天不下雨}) = 0.6$$

实际上上面的写法还是有些不方便，于是前辈又发明了**随机变量**这样的概念，把随机事件抽象成随机变量的一种取值。于是我们用一个随机变量 X 表示明天是否下雨，$X = 0$ 表示不下雨，$X = 1$ 表示下雨。于是概率又被重新定义为

$$P(X = 0) = 0.4$$
$$P(X = 1) = 0.6$$

实际上通过这样的定义，概率的表示变成了一种**映射**，它将随机变量的取值和概率值一一对应，成为这个空间中的测度。由于这个测度空间具有很多与欧氏空间相同的良好性质，很多数学分析相关的知识都可以用在这类空间上。如果继续细分，上面这个例子中的随机变量被称为**离散随机变量**。还有一种变量被称为**连续随机变量**，比如说明天下雨的降雨量这个随机变量，它取值的范围是 $[0, +\infty]$。这时很多概率相关的运算就需要用上微积分的知识了。

仅供非商业用途或交流学习使用

随机变量毕竟是随机变量，我们无法预知它的结果，但是如果已经知道了它的概率分布，猜测它可能的结果还是有可能的，这就引出了猜测它结果的一些描述量，例如**期望** E 和**方差** Var。

离散随机变量的期望

$$E(x) = \sum_x P(x)x$$

连续随机变量的期望

$$E(x) = \int_x P(x)x\mathrm{d}x$$

离散随机变量的方差

$$\mathrm{Var}(x) = \sum_x P(x)(x - E(x))^2$$

连续随机变量的方差

$$\mathrm{Var}(x) = \int_x P(x)(x - E(x))^2\mathrm{d}x$$

在很多场景下，这两个描述量很好地刻画了随机变量的性质，因此它们也被广泛使用。

聊过了单一随机变量的一些定义，下面将进入多随机变量的概率度量刻画问题。对于两个随机变量 X, Y，它们同时取值的概率被称为**联合概率** $P(X, Y)$。当然在实际运算过程中，并不需要两个随机变量在真实世界的同一时刻发生才能知道结果，只需要计算它们同时出现的可能性即可，比如说 X 表示天要不要下雨，Y 表示要不要带伞。那么 $P(X = 1, Y = 1)$ 就表示天下雨且带了伞的概率。

之所以强调两个随机变量同时取值，是因为有时两个随机变量的联合概率并不等于两件事情单独发生概率的乘积，即

$$P(X, Y) \neq P(X) \cdot P(Y)$$

原因在于随机变量之间存在着某种联系。比如说天下了雨，那么人就容易产生带伞的行为，所以两件事情同时发生实际上会包含这层含义。但是把两件事情的概率单独考虑就没有这层意思了，所以对概率的度量就会产生偏差。为了解决这个问题，**条件概率**应运而生。$P(Y|X)$ 表示当 X 取值之后 Y 取值的概率，所以联合概率正确的展开

仅供非商业用途或交流学习使用

方式是

$$P(X,Y) = P(X) \cdot P(Y|X) = P(Y) \cdot P(X|Y)$$

上面这个公式又可以推导出机器学习界最重要的概率公式（没有之一）——**贝叶斯定理**：

$$P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

这个重要的公式相信读者都有所了解。这个公式中的每一项还有自己的名字，Y 被称为隐含变量，X 被称为观察变量，$P(Y)$ 被称为**先验**（Prior），它表示我们对一个随机变量概率最初的认识；$P(X|Y)$ 被称作**似然**（Likelihood），它表示在承认先验的条件下另一个与之相关的随机变量的表现；$P(Y|X)$ 被称作**后验**（Posterior），表示当拥有 X 这个条件后 Y 的概率，由于有 X 这个条件，后验概率可能与先验概率不同；$P(X)$ 是一个**标准化常量**（Normalized Constant），由于公式中一般认为 X 已知，所以它一般被当成一个常量看待。

除了计算两个随机变量取值的概率，有些场景还需要计算两个随机变量的相关关系，这个关系可以用**协方差**表示，它的公式为

$$\begin{aligned}\mathrm{Cov}(X,Y) &= E[(X - E[X])(Y - E(Y))] \\ &= E[XY] - 2E[Y]E[X] + E[X]E[Y] \\ &= E[XY] - E[X]E[Y]\end{aligned}$$

如果 X 和 Y 的期望为 0，那么协方差就等于第一项的内容。

上面介绍了随机变量的基本表达形式和基本运算方式，但是对于如何用函数具体表达一个或多个随机变量，这个问题还是充满了挑战。为了让大家方便地分析这些随机变量，找出其中的规律，前辈们发明了很多经典的**概率分布**，这些概率分布可以当作对同一类问题的模板，满足类似性质的随机变量都可以用这些模板帮助分析。比如说对于离散随机变量，有经典的**伯努利分布**：

```
import numpy as np
def bornulli(p):
    return 1 if np.random.rand() > p else 0
```

对于连续随机变量，经典的分布有**高斯分布**：

```
def gaussian(mu, std):
    return np.random.normal(mu, std)
```

仅供非商业用途或交流学习使用

相信大家对这些分布都不陌生。一般来说，如果一个随机变量的表现和某个概率分布的具体形式非常接近（几乎处处相等），就可以认为这个随机变量服从这个概率分布。由于这些概率分布只是一个模板，具体的参数并不固定，因此在说明分布时还要明确分布的参数。

2.3.2 最大似然估计

这些分布虽然规定了它们的形式，但是其中仍然包含一些参数，如果读者已经观察到某个随机变量采样出来的样本服从某种分布，就可以根据这些样本对分布的参数进行估计。参数估计的方法在很多机器学习算法中也都有用到，其中一类十分经典的算法被称为**最大似然法**。由于每一个样本是否出现都对应着一定的概率，而且一般来说这些样本的出现都不那么偶然，因此我们希望这个概率分布的参数能够以最高的概率产生这些样本。如果观察到的数据为 $D_1, D_2, D_3, \cdots, D_N$，那么极大似然的目标如下：

$$\max P(D_1, D_2, D_3, \cdots, D_N)$$

计算联合概率总归不是一件很容易的事，如果样本数量非常大，那么对联合概率的计算会让人崩溃。所以这里一般会引入一个假设，也就是常说的**独立同分布，independent and identically distributed (i.i.d.)**。如果每一个样本既属于我们现在要求解的分布，同时它们彼此之间又相互独立，彼此出现的概率互不影响，那么根据条件独立的原则，目标公式就可以变为

$$\max \prod_{i}^{N} P(D_i)$$

看上去计算复杂度降低了许多，对于优化问题，最容易想到的方法就是求导数取极值。如果目标函数是一个凸函数，那么它的导数为 0 的点就是极值点。但是现在的公式是连乘式，求导十分麻烦，这时对数函数就派上了用场。将函数取对数，函数的极值点不会改变，于是公式就变为

$$\max \sum_{i}^{N} \log P(D_i)$$

公式变成了这个样子，求导也变得简单了许多，下面的计算也就方便了。

下面将举两个最大似然法计算的例子。首先介绍伯努利分布下随机变量的最大似然计算方法，假设 $P(X=1)=p$，$P(X=0)=1-p$，综合起来就有

仅供非商业用途或交流学习使用

$$P(X) = p^X(1-p)^{1-X}$$

如果有一组数据 D，这一组数据是从这个随机变量中采样得来的，那么就有

$$\begin{aligned}\max_p \log P(D) &= \max_p \log \prod_i^N P(D_i) \\ &= \max_p \sum_i^N \log P(D_i) \\ &= \max_p \sum_i^N [D_i \log p + (1-D_i)\log(1-p)]\end{aligned}$$

对这个式子求导，就有

$$\nabla_p \max_p \log P(D) = \sum_i^N [D_i \frac{1}{p} + (1-D_i)\frac{1}{p-1}]$$

令导数为 0，就有

$$\begin{aligned}&\sum_i^N [D_i \frac{1}{p} + (1-D_i)\frac{1}{p-1}] = 0 \\ &\sum_i^N [D_i(p-1) + (1-D_i)p] = 0 \\ &\sum_i^N (p - D_i) = 0 \\ &p = \frac{1}{N}\sum_i D_i\end{aligned}$$

这就是伯努利分布下最大似然法求出的结果，结果相当于所有采样值的平均值。

其次是基于高斯分布的最大似然法计算，推导过程也比较类似，有

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

那么用同样的方式计算，有

$$\max \log P(D) = \max_p \log \prod_i^N P(D_i)$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= \max \sum_{i}^{N} \log P(D_i) \\
&= \max \sum_{i}^{N} [-\frac{1}{2} \log(2\pi\sigma^2) - \frac{(D_i - \mu)^2}{2\sigma^2}] \\
&= \max [-\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i}^{N} (D_i - \mu)^2]
\end{aligned}
$$

首先对 μ 求导，有

$$
\frac{\partial \max_{\mu} \log P(D)}{\partial \mu} = -\frac{1}{\sigma^2} \sum_{i}^{N} (\mu - D_i)
$$

令导数为 0，就有

$$
\begin{aligned}
-\frac{1}{\sigma^2} \sum_{i}^{N} (\mu - D_i) &= 0 \\
\mu &= \frac{1}{N} \sum_{i}^{N} D_i
\end{aligned}
$$

其次对 σ^2 求导，有

$$
\frac{\partial \max_{\sigma^2} \log P(D)}{\partial \sigma^2} = -\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{i}^{N} (D_i - \mu)^2
$$

令导数为 0，有

$$
\begin{aligned}
-\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{i}^{N} (D_i - \mu)^2 &= 0 \\
\sigma^2 &= \frac{1}{N} \sum_{i}^{N} (D_i - \mu)^2
\end{aligned}
$$

从伯努利分布和高斯分布的最大似然法结果来看，它们最终求得的参数结果和期望方差的计算方式完全一致。

有关概率论的内容就介绍这些，更多内容将随着后面章节的内容介绍。

2.4 重要性采样

重要性采样（Importance Sampling）是统计中的一种采样方法。在强化学习中经常

仅供非商业用途或交流学习使用

用到这种采样方法，它主要用在一些难以直接采样的数据分布上。采样是统计中常见的一种手法，有时我们需要从某个分布中采样一些数据，并利用这些数据完成更多的运算。例如，假设有一个很复杂的概率密度函数 $p(x)$，求解随机变量基于此概率下的某个函数期望，形式化的表示就是

$$E_{x\sim p(x)}[f(x)]$$

求解它至少有两种方法。其中一种就是采用解析的方法，将上面的公式展开变成积分，然后将积分求解出来：

$$\int_x p(x)f(x)\mathrm{d}x$$

对于简单的分布形式，我们当然可以用这种方法，通过使用学到的微积分的知识直接得到结果，但如果函数 $f(x)$ 比较复杂，直接求解变得不那么直接，这种方法就不太可行了。

另一种方法采用蒙特卡罗法，这算是一种“曲线救国”的方法。我们曾经学习过大数定理，当采样数量足够大时，采样的样本就可以无限近似地表示原分布，基于这样的思想，我们的公式就变为

$$\frac{1}{N}\sum_{x_i\sim p(x),i=1}^{N} f(x_i)$$

像 numpy 这样的函数库也为我们提供了一些基本的采样函数，比如 np.random.normal()，它可以从正态分布中采样数据。采样得到的数据可以用于计算函数值，这样问题同样可以得到比较好的解决。最终结果的精确度取决于采样的数量，采样数量越多，结果的近似效果就越好，这也符合大数定理的解释。

当我们想从这些较常见的分布进行采样时，上面的函数就可以帮助我们。但是这个世界是多样的，我们总会遇到一些奇怪的概率分布，那时我们该如何进行采样呢？如果概率密度函数没有具体的公式形式呢？比如说概率函数由一个深度神经网络构成，我们只能通过输入给定的样本得到它的概率，但显然很难写出分布的具体表达式，更别说直接使用它进行采样了。

这时重要性采样就派上了用场。我们虽然无法从这个分布函数采样，但我们还有其他常见的、可以采样的分布，我们能不能对上面的公式进行一些变换，使用常见的分布采样呢？我们令待采样的分布为 $p(x)$，另一个简单可采样且定义域与 $p(x)$ 相同的概率密度函数为 $\tilde{p}(x)$，我们可以得到

$$E_{x\sim p(x)}[f(x)] = \int_x p(x)f(x)\mathrm{d}x$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= \int_x \tilde{p}(x)\frac{p(x)}{\tilde{p}(x)}f(x)\mathrm{d}x \\
&= E_{x\sim\tilde{p}(x)}[\frac{p(x)}{\tilde{p}(x)}f(x)] \\
&\simeq \frac{1}{N}\sum_{x_i\sim\tilde{p}(x),i=1}^{N}\frac{p(x)}{\tilde{p}(x)}f(x)
\end{aligned}
$$

此时我们发现，公式变成了类似上一个方法的形式，而且我们只需要从这个简单分布 $\tilde{p}(x)$ 中采样，然后分别计算样本在两个分布中的概率和函数值，最后将三者组合起来就可以得到结果。

当然，我们也不能太乐观，毕竟天下没有免费的午餐，为了检验这个方法的效果，我们将用两个常见的分布进行实验。我们假设要采样的数据来自均值为 1，标准差为 1 的高斯分布，希望用另一个高斯分布近似这个分布。这里选取了三个分布：均值都为 1，但标准差分别为 1、0.5 和 2 的高斯分布。为了更好地看出重要性采样的效果，这里的函数将选择一个比较简单的形式：$f(x) = x$。

接下来，我们分别使用三个分布近似原始分布，共完成三组实验。每一组实验中取出一个分布进行重要性采样的计算，共进行十轮计算。每一轮计算进行 100000 次采样，并将所有的采样值直接加和。最终将十轮计算结果聚合统计，得到它们的均值和标准差，然后对比三组实验的结果。对应的代码如下所示：

```
import numpy as np
import math

def gaussian(x, u, sigma):
    return math.exp(-(x - u) ** 2 / (2 * sigma * sigma)) / math.sqrt(2 *
    math.pi * sigma * sigma)

def importance_sampling_test(ori_sigma, sample_sigma):
    origin = []
    for n in range(10):
        sum = 0
        for i in range(100000):
            a = np.random.normal(1.0, ori_sigma)
            sum += a
        origin.append(sum)

    isample = []
```

仅供非商业用途或交流学习使用


```
    for n in range(10):
        sum2 = 0
        for i in range(100000):
            a= np.random.normal(1.0, sample_sigma)
            ua = gaussian(a, 1.0, sample_sigma)
            na = gaussian(a, 1.0, ori_sigma)
            sum2 += a * na / ua
        isample.append(sum2)

    origin = np.array(origin)
    isample = np.array(isample)
    print np.mean(origin), np.std(origin)
    print np.mean(isample), np.std(isample)

importance_sampling_test(1.0, 1.0)
importance_sampling_test(1.0, 0.5)
importance_sampling_test(1.0, 2)
```

在揭晓结果前，我们先来看三个分布的图像，如图 2-3 所示。

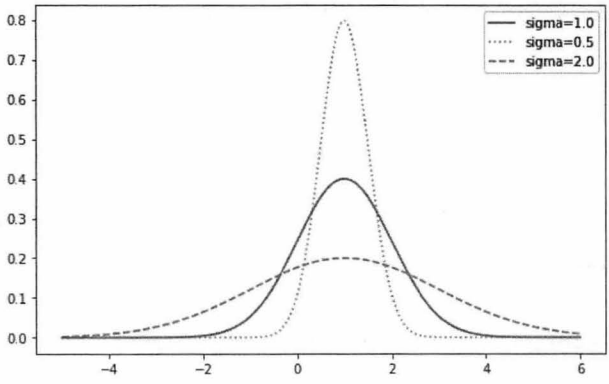


图 2-3　三个高斯分布的图像

```
    xs = np.linspace(-5, 6, 301)
    y1 = [gaussian(x, 1.0, 1.0) for x in xs]
    y2 = [gaussian(x, 1.0, 0.5) for x in xs]
    y3 = [gaussian(x, 1.0, 2) for x in xs]

    fig = plt.figure(figsize=(8,5))
```

仅供非商业用途或交流学习使用


```
plt.plot(xs, y1, label='sigma=1.0')
plt.plot(xs, y2, label='sigma=0.5', linestyle=':')
plt.plot(xs, y3, label='sigma=2.0', linestyle='--')
plt.legend()
```

可以看出，为了实现 σ 等于 1.0 的分布的采样，我们更应该使用 σ 等于 1.0 和 2.0 的分布，因为它们在取值空间上的概率分布更均匀，概率较大的区域可以覆盖原始分布概率较大的区域，不会出现因为某些区域概率较小而采样不到的情况，计算结果也相对更精准；而 σ 等于 0.5 时更有可能出现采样不均匀的情况，结果也会差一些。

基于这个分析，最终结果如下：

```
sigma=0.5, mean=99493.351966, std=4413.97863328
sigma=1.0, mean=99966.9478776, std=337.325996061
sigma=2, mean=100008.760019, std=342.846597763
```

可以看出，虽然三者在结果上存在一定差距，σ 等于 0.5 的分布在均值上离理想值稍远，另外两个分布更接近；更重要的是，σ 等于 0.5 的分布标准在标准差上和另外两个分布存在明显差异，这也说明选择一个合适的分布对重要性采样的重要性：要选择与原始分布尽可能接近的近似分布进行采样。如果选择不当，最终结果不会很好。后面我们还会接触一些利用重要性采样解决问题的场景，请读者记住决定重要性采样效果的关键因素，在后面的应用中细细体会。

2.5 信息论基础

信息论也是一门和机器学习紧密相关的学科，这里面的一些基本概念，例如熵，读者应该多少有些了解。下面笔者就来介绍熵的那些事。

大家都知道这个世界上有很多不确定的事情，比如说明天的天气，足球世界杯哪支队伍摘得桂冠。如果有人能将这些不确定的事情确定下来，或者提前告知人们它未来的状态，那么很显然这个人的能力是非常强大的。这里可以把这种描述不确定性事物的断言称为一种信息，事实上，描述任何一种不确定的事物都可能产生信息，那么这些信息的价值有没有什么差别呢？很多谍战片都在讲述地下特工如何冒着生命危险将一个机密情报传递给后方的军队，可见这个情报是非常有价值的，而敌方军队是否会不战而降呢？这个问题似乎不用考虑，大家都能给出结论。

可见不同的信息拥有不同的价值，那么它的价值该如何体现呢？从信息论的角度来说，如果一个信息的随机性越大，那么把它确定下来的价值也就越大。例如，我们有

仅供非商业用途或交流学习使用

一枚正常的硬币，一面是字，另一面是花，将硬币抛在空中然后落下，朝上的那一面会是哪一面呢？由于这里提到的是一枚正常的硬币，于是可以认为每一面朝上的概率是相同的，都是 50%；如果另一枚硬币两面都是字，那么字的一面朝上的概率就是 100%。所以猜中第二枚硬币朝上的那面比第一枚要容易得多。

虽然定性地猜测“哪枚硬币朝上一面的信息量大”这件事并不难，但是这个世界上有许多类似的事情，如何从信息论的角度比较任意两个随机变量的信息价值呢？这就需要一种定量的分析方法。于是前辈发明了“**熵**”这个概念。

首先来看看离散随机变量的“熵”。有些著作中将“熵”形容为了解真相的“**惊喜度**”。如果一个随机事件发生的概率为 100%，那么它的发生对我们来说毫无惊喜可言——因为它肯定会发生；而如果它发生的概率为百万分之一，那么它的发生一定会让人惊喜不已。可以想象百万分之一概率发生的都是什么事件——比如说彩票中奖，这样的事件让人惊喜的可能性当然非常大。

这样就把一个离散随机变量拆解成有限个或者无限个随机事件，分别把这些事件的“惊喜度”计算出来，然后把惊喜度结合起来，就是这个随机变量的惊喜度了。看上去这里需要对“惊喜度”做更具体的定义。同其他公理性质的概念一样，“惊喜度”的计算公式不会被直接给出，给出的是“惊喜度”公式应该具备的一些性质，这一点和勒贝格测度的定义方法有些类似。令这个“惊喜度”的计算公式为 $f(p(x))$，其中的输入 $p(x)$ 为某个随机事件发生的概率，$p(x) \in [0,1]$。它需要满足下面的性质：

（1）$f(1)=0$。一定发生的事情没有“惊喜”。

（2）如果 $p(x)<p(y)$，那么 $f(p(x))>f(p(y))$。换句话说，一个事情的概率越小，它的“惊喜度”越大。

（3）函数 f 在输入空间是连续的。

（4）两个事件同时发生的“惊喜度”等于各自事件发生的“惊喜度”的和，由于两个事件是互不相交的，所以这条定理和测度的次可加性性质很像。

（5）$f(0.5)=1$，这个性质主要是做归一化用的，就像外测度对测度定义的限制一样。

那么，什么样的函数能同时满足这些条件呢？科研人员最终找到了一个函数 $f(x)=-\log_2 x$。它完美地满足所有的性质。最后把随机事件发生的概率当作这个事件的权重，把所有事件的“惊喜度”进行加权平均，就得到了随机变量整体的“惊喜度”

$$\text{entropy}=-\sum_x p(x)\log p(x)$$

同样地，熵的定义也可以推广到连续变量上：

仅供非商业用途或交流学习使用

$$\text{entropy} = -\int_x p(x)\log p(x)\mathrm{d}x$$

这个公式虽然满足上面的那些性质，但是看上去总是不够直观，回到刚才那个硬币的例子，假设手里有一枚硬币，硬币有字一面朝上的概率为 p，所以花一面朝上的概率为 $1-p$。知道这些就可以写出求这个问题熵的公式

$$\text{entropy}(p) = -p\log p - (1-p)\log(1-p)$$

这个公式的图像可以通过下面的代码生成：

```
import numpy as np
x = np.linspace(0,1,101)
y = -x * np.log2(x) -(1-x) * np.log2(1-x)
y[np.isnan(y)]=0
plt.plot(x,y)
plt.show()
```

结果如图 2-4 所示。

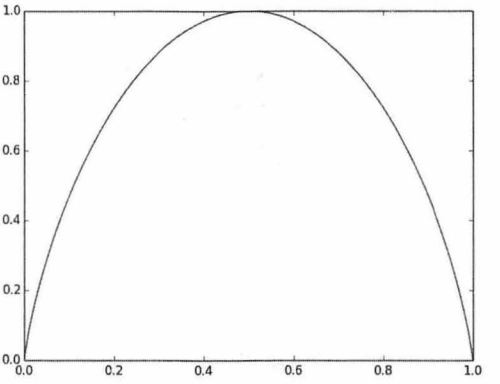


图 2-4 硬币问题的概率与熵的关系图

当两个面朝上的概率相同时，它的熵最大。这也和我们的直觉相符，正是因为两个事件的概率相同，我们才难以判断最终结果，这时知道结果的“总体惊喜度”才会最大。

2.6 KL 散度

KL 散度是概率论和信息论中十分重要的一个概念。它是描述两个概率分布 P 和 Q 差异的一种方法。

仅供非商业用途或交流学习使用

离散概率分布的 KL 散度计算公式为

$$\mathrm{KL}(p\|q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

连续概率分布的 KL 散度计算公式为

$$\mathrm{KL}(p\|q) = \int p(x) \log \frac{p(x)}{q(x)} \mathrm{d}x$$

KL 散度可以很好地测量两个概率分布之间的距离。两个分布越接近，KL 散度越小；两个分布越远，KL 散度就越大。

KL 散度的结果是非负的，这里可以简单证明得出

$$\begin{aligned}\mathrm{KL}(p\|q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= -\sum_x p(x) \log \frac{q(x)}{p(x)}\end{aligned}$$

由于对数函数是一个上凸函数，所以有

$$\begin{aligned}&\geqslant -\log[\sum_x p(x) \frac{q(x)}{p(x)}] \\ &= -\log[\sum_x q(x)] \\ &= -\log 1 = 0\end{aligned}$$

由此得证。

看完了 KL 散度的基本性质，下面来看一个实际的案例：假设有两个随机变量 x_1 和 x_2，各自服从一个高斯分布 $N_1(\mu_1, \sigma_1^2), N_2(\mu_2, \sigma_2^2)$，那么这两个分布的 KL 散度该怎么计算呢？

在前面的 2.3 节中，读者知道了高斯分布的概率密度函数：

$$N(\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{\frac{(x-\mu)^2}{2\sigma^2}}$$

那么 KL(p1‖p2) 就等于

$$\int p_1(x) \log \frac{p_1(x)}{p_2(x)} \mathrm{d}x$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= \int p_1(x)(\log p_1(x) - \log p_2(x))\mathrm{d}x \\
&= \int p_1(x)(\log \frac{1}{\sqrt{2\pi\sigma_1^2}}\mathrm{e}^{\frac{(x-\mu_1)^2}{2\sigma_1^2}} - \log \frac{1}{\sqrt{2\pi\sigma_2^2}}\mathrm{e}^{\frac{(x-\mu_2)^2}{2\sigma_2^2}})\mathrm{d}x \\
&= \int p_1(x)(-\frac{1}{2}\log 2\pi - \log\sigma_1 - \frac{(x-\mu_1)^2}{2\sigma_1^2} + \frac{1}{2}\log 2\pi + \log\sigma_2 + \frac{(x-\mu_2)^2}{2\sigma_2^2})\mathrm{d}x \\
&= \int p_1(x)(\log\frac{\sigma_2}{\sigma_1} + [\frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2}])\mathrm{d}x \\
&= \int (\log\frac{\sigma_2}{\sigma_1})p_1(x)\mathrm{d}x + \int (\frac{(x-\mu_2)^2}{2\sigma_2^2})p_1(x)\mathrm{d}x - \int (\frac{(x-\mu_1)^2}{2\sigma_1^2})p_1(x)\mathrm{d}x \\
&= \log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\int ((x-\mu_2)^2)p_1(x)\mathrm{d}x - \frac{1}{2\sigma_1^2}\int ((x-\mu_1)^2)p_1(x)\mathrm{d}x
\end{aligned}
$$

到这一步，公式的最后一项可以化简，这个积分符号里面的东西是不是看着很熟悉？没错，它就是连续变量方差的计算公式，于是经过约分，这一项就变成了 $\frac{1}{2}$。继续推导，有

$$
\begin{aligned}
&= \log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\int ((x-\mu_2)^2)p_1(x)\mathrm{d}x - \frac{1}{2} \\
&= \log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\int ((x-\mu_1+\mu_1-\mu_2)^2)p_1(x)\mathrm{d}x - \frac{1}{2} \\
&= \log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\Big[\int (x-\mu_1)^2 p_1(x)\mathrm{d}x + \int (\mu_1-\mu_2)^2 p_1(x)\mathrm{d}x \\
&\qquad\qquad +2\int (x-\mu_1)(\mu_1-\mu_2)p_1(x)\mathrm{d}x\Big] - \frac{1}{2} \\
&= \log\frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2}\Big[\int (x-\mu_1)^2 p_1(x)\mathrm{d}x + (\mu_1-\mu_2)^2\Big] - \frac{1}{2} \\
&= \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}
\end{aligned}
$$

经过大段让人发懵的公式推导，最终的结论已经得出。假设 N_2 是一个正态分布，也就是说 $\mu_2 = 0, \sigma_2^2 = 1$，那么 N_1 长成什么样子能够让 KL 散度尽可能地小呢？

将 N_1 中的变量代入，公式变成

$$
\mathrm{KL}(\mu_1, \sigma_1) = -\log\sigma_1 + \frac{\sigma_1^2 + \mu_1^2}{2} - \frac{1}{2}
$$

读者简单分析一下就能猜测到，当 $\mu_1 = 0, \sigma_1 = 1$ 时，KL 散度最小。从公式中可以看出，如果 μ_1 偏离了 0，那么 KL 散度一定会变大；而 σ_1 的变化对 KL 散度的影响则不那么明显：

仅供非商业用途或交流学习使用

- 当 σ_1 大于 1 时，$\frac{1}{2}\sigma_1^2$ 将越变越大，而 $-\log\sigma_1$ 越变越小。
- 当 σ_1 小于 1 时，$\frac{1}{2}\sigma_1^2$ 将越变越小，而 $-\log\sigma_1$ 越变越大。

那么，哪边的力量更强大呢？由于公式比较简单，这里就将它的图像画出来进行比较，如图 2-5 所示。

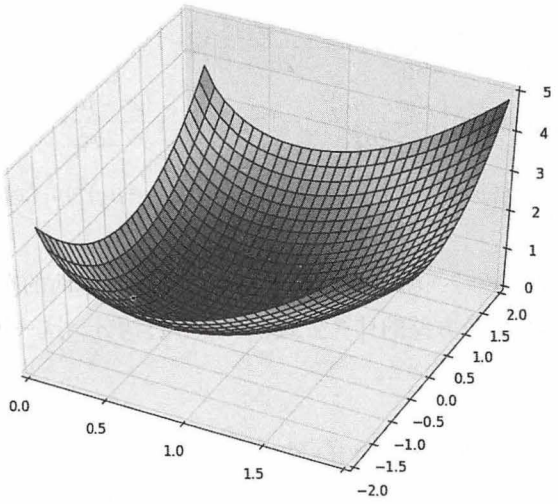


图 2-5　两个高斯分布的 KL 散度图

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
x = np.linspace(0.1,2,31)
y = np.linspace(-2,2,31)
X,Y = np.meshgrid(x,y)
Z = -np.log(X)+X*X+Y*Y/2-0.5
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap='rainbow')
plt.show()
```

从图 2-5 中可以看出二次项的威力更大，函数一直保持为非负，这和前面推导的结论（KL 散度非负的结论）是完全一致的。

仅供非商业用途或交流学习使用

2.7 凸函数及其性质

凸函数是机器学习中经常见到的一种形式。它拥有非常好的性质，在计算上拥有更多的便利。这种感觉就好像看到一个复杂的 7 次函数 $f(x)=ax^7+bx+c$，但实际上它的 7 次项系数 $a=0$，这个披着复杂函数外皮的家伙其实很好分析。虽然后面介绍的深度学习网络的优化曲面并不具备凸函数的性质，但是凸函数还是作为一个十分经典的模型被大家不断研究学习，更何况它也是众多优化算法的基础，因此下面开始介绍这类函数及其相关特点。

要介绍凸函数，首先要介绍凸集。如果一个集合 C 被称为凸集，那么这个集合中的任意两点间的线段仍然包含在集合中，如果用形式化的方法描述，那么对于任意两个点 $x_1,x_2\in C$，和任意一个处于 $[0,1]$ 的实数 θ，都有

$$\theta x_1+(1-\theta)x_2\in C$$

下面给出凸函数和非凸函数的对比图像，如图 2-6 所示。可以看出，对于左边的凸集区域，任意两点间的线段都在集合中，而对于右边的非凸集区域，我们可以找到两点间的一条线段，使得线段上的点在集合外。

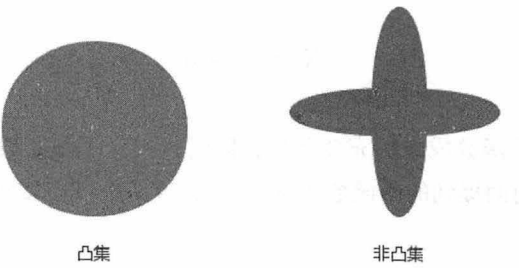


图 2-6 凸集和非凸集

凸函数的定义域就是一个凸集，除此之外，它还具备另外一个性质：给定函数中任意两点 x,y，和任意一个处于 $[0,1]$ 的实数 θ，有

$$f(\theta x+(1-\theta)y)\leqslant\theta f(x)+(1-\theta)f(y)$$

这里给出一个凸函数的例子：$f(x)=x^2$，然后看看这个性质在函数上的表现，它的图像如图 2-7 所示，图中的横截线代表不等式右边的内容，横截线下方的曲线代表不等式左边的内容，从图中确实可以看出不等式所表达的含义，如果一个凸函数像 x^2 这样是一个严格的凸函数，那么实际上除非 θ 等于 0 或者 1，否则等号不会成立。

仅供非商业用途或交流学习使用

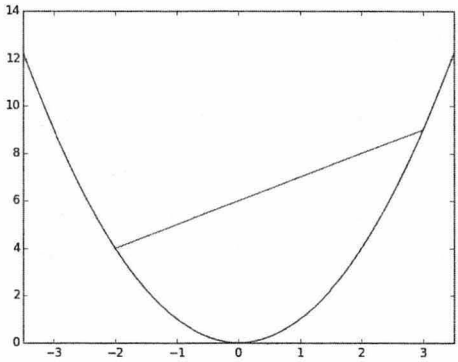


图 2-7　凸函数性质展示

凸函数满足的这个不等式也被称为 **Jensen 不等式**，要判定一个函数是不是凸函数，除了用 Jensen 不等式来判定，还可以用下面两种方式判定。首先是一阶导数条件。令 x, y 是凸函数 f 的任意两个点，那么下式成立：

$$f(y) \geqslant f(x) + \nabla f(x)^{\mathrm{T}}(y - x)$$

其次是二阶导数条件：令 x 是凸函数 f 上的任意一点，那么下式成立：

$$\nabla^2 f(x) \geqslant 0$$

这两个条件是凸函数成立的充要条件，同时也揭示了凸函数的重要性质。从直观的角度分析，对于前面提到的凸函数 $f(x) = x^2$，这两个性质显然成立：

$$\begin{aligned} f(y) = y^2 &\geqslant x^2 + 2x(y - x) \\ &= x^2 + 2x(y - x) + (y - x)^2 - (y - x)^2 \\ &= [x + (y - x)]^2 - (y - x)^2 \\ &= y^2 - (y - x)^2 \\ \nabla^2 f(x) &= 2 \geqslant 0 \end{aligned}$$

凸函数究竟有什么良好的性质，值得读者花大力研究呢？

如果一个点是凸函数的局部最优值，那么这个点就是函数的全局最优值。更进一步，如果这个函数是强凸函数，那么这个点是函数唯一的全局最优值。这个问题可以通过一阶条件证明，当一个点 x 是凸函数的局部最优值时，它的导数为 0，那么对于任意

仅供非商业用途或交流学习使用

的点 y，都有

$$f(y) \geqslant f(x) + \nabla f(x)^{\mathrm{T}}(y - x) = ff(x)$$

当函数为强凸函数时，大于等于号将变为大于号。

有了这个性质，我们就可以放心大胆地做函数的优化了。只要找到一个导数为 0 的局部最优值，我们就找到了全局最优值。凸函数相关的理论还有很多，本节就介绍到这里。

2.8 机器学习的基本概念

在经典的机器学习入门书籍中，读者一般会学到很多经典的机器学习模型，由于本书介绍的问题主要是监督学习，所以主要介绍一些监督学习相关的内容。通过前面的介绍，读者已经对模型这个概念有了一定的了解，也了解了模型和函数的关系。**在很多场景下，模型都是由函数构成的**，函数的形式有很多种，有的比较复杂，有的比较简单。下面假设有为模型准备好作为输入的特征数据 X 和作为输出的结果数据 Y，针对不同的任务，监督学习又可以分成几个具体的类别。

首先是输出 Y 的形式，如果结果 Y 是一个连续的变量，那么这个问题通常被称为**回归问题**。这里可以用 Python 的 scikit-learn 工具包随机生成一个可以可视化的回归问题数据集。

```
import numpy
import sklearn.datasets as d
import matplotlib.pyplot as plt
reg_data = d.make_regression(100, 1, 1, 1, 1.0)
plt.plot(reg_data[0], reg_data[1])
plt.show()
```

这个例子相对来说十分简单，它的结果也十分直观，输入数据和输出数据都是 1 维的，它们的关系如图 2-8 所示。

如果输出 Y 的形式是离散的变量，例如上面提到的手写数字分类问题，那么这个问题就是分类问题。这里也给出一个简单的分类问题的数据集。

```
import numpy
import sklearn.datasets as d
import matplotlib.pyplot as plt
cls_data = d.make_classification(100, 2, 2, 0, 0, 2)
```

仅供非商业用途或交流学习使用

这个例子对应的图像也十分简单，如图 2-9 所示。

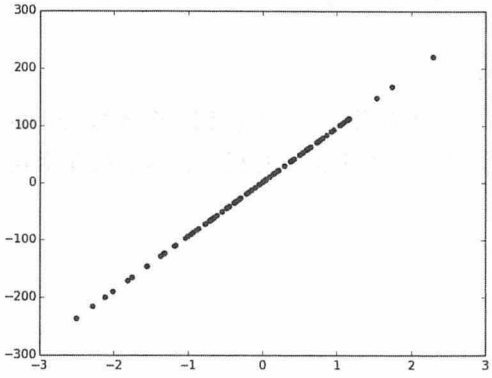


图 2-8　一个回归问题的数据展示

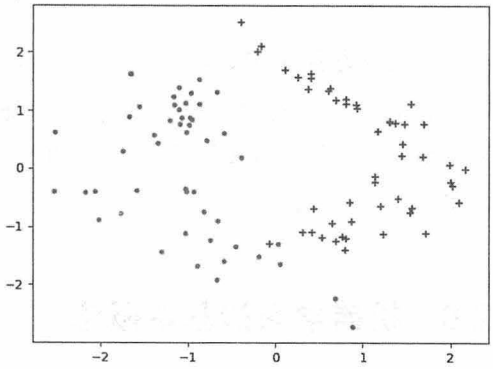


图 2-9　分类问题数据示例

分类问题的目标就是让模型把点号和加号的点区分开。

上面看过了问题的形式，下面来看看问题的解决方法。这里以回归问题为例，为了解决问题，我们可以建立一个模型，让模型充当输入和输出的桥梁。例如，对于上面的回归问题，可以建立线性模型

$$y = wx + b$$

其中 x 和 y 分别是模型的输入和输出，w 和 b 是模型的两个参数，这两个参数在模型学习训练前是不固定的，经过训练后，它才会确定下来。

第一步明确了模型，下面就来看看模型的**目标函数**。这里采用平方损失函数，如果数据的真实输出为 t，模型的输出为 y，那么损失函数定义为

$$\text{Loss} = \frac{1}{2}(t - y)^2$$

平方损失函数很好地度量了真实输出和模型输出之间的距离。如果模型输出和真实输出差距较大，那么模型损失函数的结果也较大，说明了模型还有很大的改进空间。从损失函数的定义来看，损失函数的最小值为 0，这就是模型要努力的方向。所以，最终函数的目标是最小化损失函数：

$$\min_{w,b} \text{Loss} = \min_{w,b} \frac{1}{2}(t - y)^2$$

完成了模型和目标函数的定义，下面就来看看模型如何学习。对于这个简单的问题，直接求出它的闭式解是完全可行的。但在大多数真实场景中，闭式求解模型最优值

仅供非商业用途或交流学习使用

是不现实的，因为那将花费非常长的时间。通常，我们用基于梯度的方式采用优化的方法求解，因此这个过程也被称作**优化**。优化的基本流程分成 3 个步骤：

（1）为参数设定一些初始值。

（2）利用梯度信息计算参数的更新量。

（3）判断参数优化是否完成，如果完成，则停止优化，否则回到第 2 步。

这 3 个步骤中，第 3 步的内容相对简单，一般有以下几种判断方法。

（1）判断参数所在位置的梯度是否靠近 0（依梯度收敛）。

（2）判断参数的更新量是否靠近 0（柯西列收敛）。

（3）判断预设的迭代轮数是否已经完成。

由于机器学习中的模型一般是定义良好的函数，因此这一部分受到的关注比较小。此外，对于凸函数优化来说，第 1 步的内容也比较简单，优化的重头戏全部在第 2 步中。

对深度学习来说，由于种种原因，越来越多的人发现了第 1 步的重要性。本节只涉及机器学习中的这几个概念，更多的概念将在后面章节中慢慢介绍。通过本节，希望读者至少对模型、目标函数和优化这 3 个概念有一些了解。

2.9 机器学习的目标函数

前面介绍了机器学习中常见的组成部分，下面要花一点时间介绍损失函数的内容。常见的损失函数有**平方损失函数**和**交叉熵损失函数**，这两种损失函数有什么区别和联系呢？

平方损失函数是一种比较容易理解的损失函数，它直接衡量了模型输出和标准答案在数值上的差距。定义模型输出的向量为 $\boldsymbol{y}$，训练数据的标签向量为 $\boldsymbol{t}$，那么平方损失函数的定义为

$$\text{SquareLoss} = \frac{1}{2}(\boldsymbol{y} - \boldsymbol{t})^2$$

交叉熵的定义就没那么直接了，它需要用到 2.5 节中信息论的知识。这里简单回顾一下，熵这个概念衡量了一个随机变量带来的“惊喜度”，它通过计算每一个取值的惊喜度汇总得到。实际上，这个公式的计算涉及两个部分——惊喜度 + 汇总。如果惊喜度的计算使用一个概率分布，而汇总使用另一个概率分布，那么结果会变成什么样子呢？这就是交叉熵要表达的内容。

举一个简单的例子，有两个服从伯努利分布的随机变量 P 和 Q，它们的交叉熵 $H(P,Q)$ 为

$$H(P,Q) = -P(0)\log Q(0) - (1 - P(0))\log(1 - Q(0))$$

仅供非商业用途或交流学习使用

由于这个公式只有 2 个参数，于是可以将它的图像画出来：

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.linspace(0.01,0.99,101)
Y = np.linspace(0.01,0.99,101)
X,Y = np.meshgrid(X,Y)
Z = -X * np.log2(Y) - (1-X)* np.log2(1 - Y)
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap='rainbow')
plt.show()
```

生成的图像如图 2-10 所示。

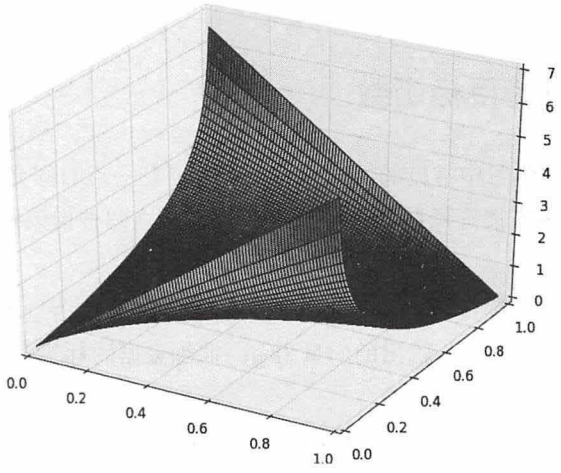


图 2-10　伯努利分布下两个随机变量的交叉熵

当两个分布的取值完全相同时，交叉熵的取值最小。对于损失函数来说，当然是希望模型输出和数据标签相同，因此损失函数的目标就是让交叉熵尽可能地变小。在这一点上函数和目标是一致的。

看上去两个损失函数的目标是差不多的，那么该如何应用它们呢？一般来说，如果最终输出的结果是回归问题的一个连续型变量，使用平方损失函数更合适；如果最终输出是分类问题的一个离散 Ont-Hot 向量，那么交叉熵损失函数更合适。

仅供非商业用途或交流学习使用

但是，看上去平方损失函数既可以做回归问题的损失函数，又可以做分类问题的损失函数，因为它们都可以用距离来衡量，那么平方损失函数在做分类问题的损失函数时有什么弱点呢？

这个问题可以从两个方面回答。一方面，直观上看，平方损失函数对每一个输出结果都十分看重，而交叉熵损失函数只看重正确分类的结果。假设我们遇到了一个三分类问题，模型的输出自然是一个三维的实数向量 (a,b,c)，向量的每一个元素都表示了对这个类别的预测概率，假设数据的真实结果为 (1,0,0)，那么两个损失函数的公式为

$$\text{SquareLoss} = (a-1)^2 + (b-0)^2 + (c-0)^2 = (a-1)^2 + b^2 + c^2$$

$$\text{CrossEntropyLoss} = -1 \times \log a - 0 \times \log b - 0 \times \log c = -\log a$$

可以看出，平方损失函数考虑的内容实际上比交叉熵损失函数多。也就是说，在模型优化的过程中，交叉熵损失的梯度只和正确分类的预测结果有关，而平方损失的梯度还和错误分类有关。除了让正确分类尽可能变大，平方损失函数还会让错误分类都变得更平均，但实际中后面这个调整是不必要的，所以平方损失实际上完成了额外的工作，因此它的实用性就显得没那么强了。但是话说回来，这个考虑在回归问题上显得非常重要，因此在回归问题上交叉熵损失显然就不合适了。

另一方面，从理论角度分析，两个损失函数的源头不一样。平方损失函数假设最终结果都服从高斯分布，而高斯分布实际上是一个连续变量，并不是一个离散变量。如果假设结果变量服从均值为 t，方差为 σ 的高斯分布，那么利用最大似然法就可以优化它的负对数似然，公式最终变为

$$= \max \sum_{i}^{N} [-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(t_i - y)^2}{2\sigma^2}]$$

除去与 y 无关的项目，最终剩下来的公式就是平方损失函数的形式。

2.10 总结

本章和读者一同回顾了机器学习中常用的数学基础知识。

（1）线性代数基础：矩阵、线性变换、线性相关、特征值与特征向量。

（2）线性代数中的对阵矩阵。

（3）概率论基础：概率、贝叶斯公式、最大似然法估计和重要性采样。

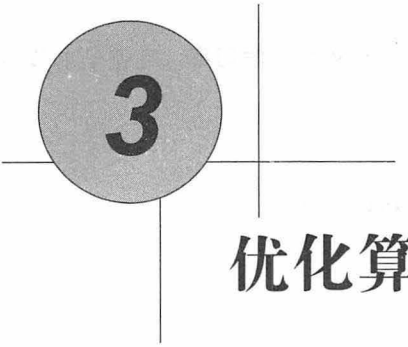
仅供非商业用途或交流学习使用

（4）信息论基础：熵和 KL 散度。

（5）凸函数：凸函数的性质。

（6）机器学习的基本概念：分类回归模型、目标函数和优化。

仅供非商业用途或交流学习使用



3 优化算法

作为机器学习中十分重要的一环，优化算法受到了极大的重视。好的优化算法意味着可以更好、更快地找到目标模型，所以优化算法相关的知识向来充满了吸引力。深层模型的发展又让它备受瞩目：深层模型的目标函数不再是凸函数，优化曲面变得更复杂，别说寻找一个优秀的优化算法，完成正常的优化都变得困难。本章将介绍三个被学术界和工业界广泛研究和使用的优化算法：梯度下降法、共轭梯度法和自然梯度法。了解这三个优化算法会对后面的学习有很大帮助。

3.1 梯度下降法

3.1.1 什么是梯度下降法

作为大众耳熟能详的优化算法，梯度下降法（Gradient Descent）受到了太多的关注。梯度下降法极易理解，但凡学过数学的读者都知道，函数的梯度方向表示了函数值增长速度最快的方向，那么和它相反的方向就可以看作函数值减少速度最快的方向。就机器学习模型优化的问题而言，当目标设定为求解目标函数最小值时，只要朝着梯度下降的方向前进，就能不断逼近最优值。

那么梯度下降算法怎么实现呢？这里给出一种最简单的梯度下降算法——固定学习率的方法，这种梯度下降算法由两个函数和三个变量组成。

- 函数 1：待优化的函数 $f(x)$，它可以根据给定的输入返回函数值。

仅供非商业用途或交流学习使用

- 函数 2：待优化函数的导数 $g(x)$，它可以根据给定的输入返回函数的导数值。
- 变量 x：保存当前优化过程中的参数值，优化开始时该变量将被初始化成某个数值，优化过程中这个变量会不断变化，直到它找到最小值。
- 变量 grad：保存变量 x 点处的梯度值。
- 变量 step：表示沿着梯度下降方向行进的步长，也被称为**学习率**。它就是本节的主角，在优化中它将固定不变。

包含上面 5 个元素的梯度下降算法如下所示。

```
def gd(x_start, step, g):    # 名称gd代表Gradient Descent
    x = x_start
    for i in range(20):
        grad = g(x)
        x -= grad * step
        print '[ Epoch {0} ] grad = {1}, x = {2}'.format(i, grad, x)
        if abs(grad) < 1e-6:
            break;
    return x
```

除了上面介绍的内容，代码中还加入了优化的终止条件。优化的目标是寻找梯度为 0 的极值点，代码在每一轮迭代结束后衡量变量 x 所在的梯度值，因此当梯度值足够小时，就认为 x 已经进入最优值附近一个极小的邻域，x 和最优值之间的差别不再明显。这时就可以停止优化。

3.1.2 优雅的步长

明确了算法，下面就通过一个例子展示梯度下降法的优化效果。这里用一个十分简单的函数作为例子。

```
def f(x):
    return x * x - 2 * x + 1


def g(x):
    return 2 * x - 2
```

这个函数 $f(x)$ 就是读者在中学常见的二次函数 $f(x) = x^2 - 2x + 1$，读者可以很轻松地看出，最小值是 $x = 1$，此时函数值为 0。为了让读者对这个函数有更直观的认识，这里将图画出来。

仅供非商业用途或交流学习使用


```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5,7,100)
y = f(x)
plt.plot(x, y)
```

这个函数如图 3-1 所示。

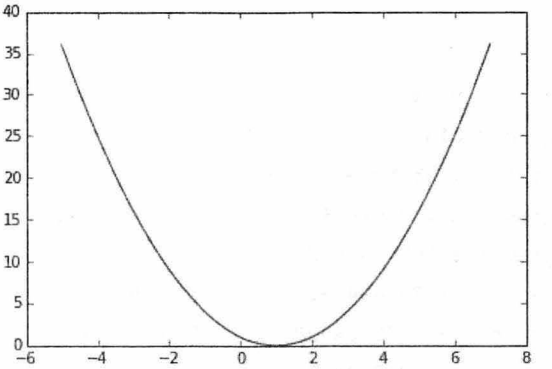


图 3-1 梯度下降法的案例函数

可以清楚地看出这个函数的图像就是一个很简单的抛物线。$x = 1$ 是函数的最小值。下面就用梯度下降法计算它的极值：

```
gd(5,0.1,g)
```

运行后得到下面的输出（结果有省略）：

```
[ Epoch 0 ] grad = 8, x = 4.2
[ Epoch 2 ] grad = 5.12, x = 3.048
[ Epoch 4 ] grad = 3.277, x = 2.311
[ Epoch 6 ] grad = 2.097, x = 1.839
[ Epoch 8 ] grad = 1.342, x = 1.537
[ Epoch 10 ] grad = 0.859, x = 1.344
[ Epoch 12 ] grad = 0.550, x = 1.220
[ Epoch 14 ] grad = 0.352, x = 1.141
[ Epoch 16 ] grad = 0.225, x = 1.090
[ Epoch 18 ] grad = 0.144, x = 1.058
```

可以看到，初始值 x 从 5 出发，梯度值在不断下降，经过 20 轮迭代，x 虽然没有完全等于 1，但是在迭代中它正不断地逼近最优值 $x = 1$。

仅供非商业用途或交流学习使用

以上就是梯度下降法的展示。这个例子留下了一点遗憾：最后 x 离最优值的距离并不算近。是不是因为每一轮迭代时的步长设置得太小，导致优化值没有更快地收敛到最优值呢？如果是这样，那么是不是增大步长就可以在 20 轮迭代内看到优化结束呢？抱着这样的想法，让我们重新进行一次优化，这一次的优化步长被设置为之前的 1000 倍（有点夸张）。

```
gd(5,100,g)
```

如下代码就是步长加大后的结果：

```
[ Epoch 0 ] grad = 8, x = -795
[ Epoch 2 ] grad = 316808, x = -31522395
[ Epoch 4 ] grad = 1.254591e+10, x = -1.248318e+12
[ Epoch 6 ] grad = 4.968307e+14, x = -4.943466e+16
[ Epoch 8 ] grad = 1.967499e+19, x = -1.957662e+21
[ Epoch 10 ] grad = 7.791494e+23, x = -7.752537e+25
[ Epoch 12 ] grad = 3.085510e+28, x = -3.070082e+30
[ Epoch 14 ] grad = 1.221893e+33, x = -1.215783e+35
[ Epoch 16 ] grad = 4.838817e+37, x = -4.814623e+39
[ Epoch 18 ] grad = 1.916220e+42, x = -1.906639e+44
```

可以看到，参数的梯度不但没有收敛，反而越来越大。优化的本意是让目标值朝着梯度下降的方向前进，结果它却走向了另外一个方向。为什么会出现这样的情况呢？为什么梯度会不降反升呢？是我们的算法本身有问题，还是梯度的设置有问题？解释这个问题还要回到梯度这个概念本身。

实际上，函数在某一点的梯度指的是它在当前变量处的梯度，对于这一点来说，它的梯度方向指向了函数上升的方向，可以利用泰勒公式证明在一定范围内，沿着负梯度方向前进，函数值是会下降的。但是，公式只能保证在一定范围内是成立的，从函数的实际图像中也可以看出，如果优化的步长太大，就有可能跳出函数值下降的范围，那么函数值是否下降就不好说了。当然有可能越变越大，造成优化的悲剧。

如何避免这种悲剧发生呢？简单的方法就是将步长减少，像前面的实验那样设得小一些。当然还有一些线搜索（Line-Search）的方法可以通过其他限定条件避免这样的事情发生，本节就不做介绍了。

既然理论没错，那么看起来只能通过修改步长来完成这个问题了。既然小步长会使目标值的梯度下降，大步长会使梯度发散，那么有没有一个步长会让优化问题原地打转呢？在这个问题中，这样的步长是存在且容易找到的。

仅供非商业用途或交流学习使用

由于梯度优化总会造成数值的改变，所以每一步优化都让目标值原地打转是不太现实的，这里可以假设目标值从 A 变为 B，再从 B 变为 A，如此循环更新。假设：从 $x=5$ 出发，经过一轮迭代，x 被更新到了另一个值 x'，再用 x' 继续迭代，x' 又更新到了 5。上述过程可以写成如下方程。

$$x=5, g(x)=8, \text{新的值 } x'=5-8\times\text{step}$$

$$g(x')=2\times(5-8\times\text{step})-2, \text{回到过去：} x'-g(x')\times\text{step}=x=5$$

合并公式求解得，$\text{step}=1$。

也就是说 $\text{step}=1$ 时，求解会原地打转，那么接下来就来试一下。

```
gd(5,1,g)
```

它的结果如下所示：

```
[ Epoch 0 ] grad = 8, x = -3
[ Epoch 1 ] grad = -8, x = 5
[ Epoch 2 ] grad = 8, x = -3
[ Epoch 3 ] grad = -8, x = 5
[ Epoch 4 ] grad = 8, x = -3
[ Epoch 5 ] grad = -8, x = 5
[ Epoch 6 ] grad = 8, x = -3
[ Epoch 7 ] grad = -8, x = 5
[ Epoch 8 ] grad = 8, x = -3
[ Epoch 9 ] grad = -8, x = 5
```

和预想的一样，目标值在函数中打转，梯度下降法失效了。

通过上面的实验可以发现，对于初始值为 5 这个点，当步长大于 1 时，梯度下降法会出现求解目标值发散的现象；而步长小于 1 时，则不会发散，参数会逐渐收敛。所以 1 就是步长的临界点。那么问题又来了，对于别的初始值，这个规律还适用吗？接下来就把初始值换成 4，再进行一次实验。

```
gd(4,1,g)
```

它的结果如下所示：

```
[ Epoch 0 ] grad = 6, x = -2
[ Epoch 1 ] grad = -6, x = 4
[ Epoch 2 ] grad = 6, x = -2
[ Epoch 3 ] grad = -6, x = 4
```

仅供非商业用途或交流学习使用


```
[ Epoch 4 ] grad = 6, x = -2
[ Epoch 5 ] grad = -6, x = 4
[ Epoch 6 ] grad = 6, x = -2
[ Epoch 7 ] grad = -6, x = 4
[ Epoch 8 ] grad = 6, x = -2
[ Epoch 9 ] grad = -6, x = 4
```

实验发现，步长等于 1 时，初始值设置成（最优值除外）任何数字，参数都不会收敛到最优值。这个实验揭示了一个道理：对于这个二次函数，如果采用固定步长的梯度下降法进行优化，步长要小于 1，否则不论初始值等于多少，问题都会发散或者原地打转！

如果再换一个函数：$4x^2 - 4x + 1$，它的安全步长是多少呢，还是 1 吗？当然不是。这一次安全步长的值是 0.25，相应的实验如下：

```
def f2(x):
    return 4 * x * x - 4 * x + 1
def g2(x):
    return 8 * x - 4
gd(5,0.25,g2)
```

它的结果如下所示：

```
[ Epoch 0 ] grad = 36, x = -4.0
[ Epoch 1 ] grad = -36.0, x = 5.0
[ Epoch 2 ] grad = 36.0, x = -4.0
[ Epoch 3 ] grad = -36.0, x = 5.0
[ Epoch 4 ] grad = 36.0, x = -4.0
[ Epoch 5 ] grad = -36.0, x = 5.0
[ Epoch 6 ] grad = 36.0, x = -4.0
[ Epoch 7 ] grad = -36.0, x = 5.0
[ Epoch 8 ] grad = 36.0, x = -4.0
[ Epoch 9 ] grad = -36.0, x = 5.0
```

实验结果和猜测完全一致。至此，相信读者对步长的设置有了更深的认识。这组实验说明了梯度下降法简单中的不简单，虽然固定步长的方法看上去简单直观，但是步长的选择要慎重。一个一维的二次函数都存在优化问题，对 CNN 网络优化来说，优化曲面要复杂得多。采用固定步长优化的方法实际上是“步步惊心”，实战中需要一定的技巧并且多尝试才能找到最合适的步长。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.2 动量算法

本节将介绍动量（Momentum）算法[1]。动量是物理课上学习过的一个概念，正如它的中文名一样，在优化求解的过程中，动量代表了前面优化过程积累的“能量”，它将在后面的优化过程中持续发威，推动目标值前进。拥有了动量，一个已经结束的更新量不会立刻消失，只会以一定的形式衰减，剩下的能量将继续在优化中发挥作用。

上面介绍的概念比较抽象，以下就是基于动量的梯度下降的代码（基于前面的梯度下降法做了一些修改）。

```
def momentum(x_start, step, g, discount = 0.7):
    x = np.array(x_start, dtype='float64')
    pre_grad = np.zeros_like(x)
    for i in range(50):
        grad = g(x)
        pre_grad = pre_grad * discount + grad * step
        x -= pre_grad

        print '[ Epoch {0} ] grad = {1}, x = {2}'.format(i, grad, x)
        if abs(sum(grad)) < 1e-6:
            break;
    return x
```

代码中多出了一个新变量 pre_grad。这个变量就是用于存储历史积累的动量，每一轮迭代动量都会乘以一个打折量（discount）做能量衰减，但是它依然会被用于更新参数。

每个事物的存在必然有它的道理，动量算法和前面介绍的梯度下降法相比有什么优点呢？用形象的话来说，它可以帮助目标值穿越“狭窄山谷”形状的优化曲面，从而到达最终的最优点。那么，什么是“山谷”，怎么理解“穿越山谷”这个词语呢？下面用一个例子来解释。这次的问题复杂些，是一个二元二次函数：$z = x^2 + 50y^2$。

```
def f(x):
    return x[0] * x[0] + 50 * x[1] * x[1]
def g(x):
    return np.array([2 * x[0], 100 * x[1]])
xi = np.linspace(-200,200,1000)
yi = np.linspace(-100,100,1000)
X,Y = np.meshgrid(xi, yi)
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
Z = X * X + 50 * Y * Y
```

函数在等高线图上的样子如图 3-2 所示。

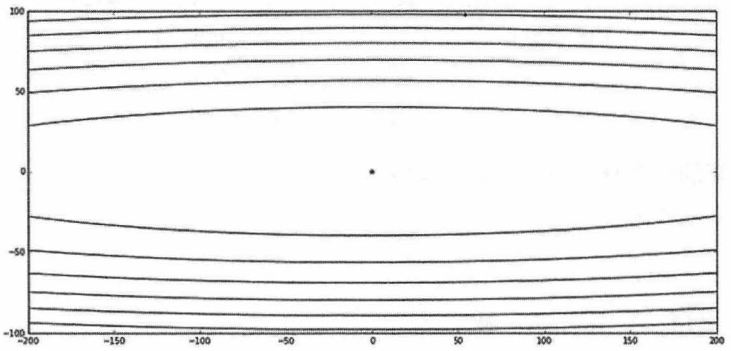


图 3-2　二次目标函数的等高线图

其中中心的点表示最优值。把等高线上的图像想象成地形图，从等高线的疏密程度可以看出，这个函数在 y 轴方向十分陡峭，在 x 轴方向则相对平缓。也就是说，函数在 y 轴的方向导数比较大，在 x 轴的方向导数比较小。图 3-2 所示的区域可以看作一个“山谷”。如果用 3.1 节介绍的固定步长的梯度下降法尝试优化，则有：

```
gd([150,75], 0.016, g)
```

将 50 轮迭代过程中参数的优化过程图画出来，如图 3-3 所示。

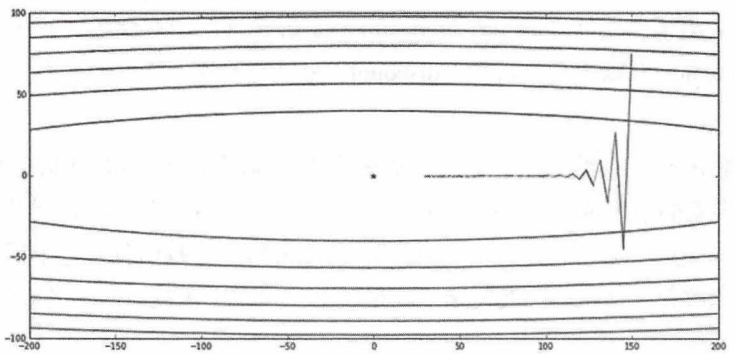


图 3-3　梯度下降法的优化曲线

可以看出目标值从某个点出发，整体趋势向着最优点前进，它和最优值的距离不断靠近，说明优化过程在收敛，步长设置是没有问题的，但是前进的速度似乎有点乏力，50 轮迭代并没有到达最优值，有可能是步长设置得偏小。有了 3.1 节的经验，这次设置步长时很谨慎。我们只将步长稍变大一点，结果如图 3-4 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
res, x_arr = gd([150,75], 0.019, g)
contour(X,Y,Z, x_arr)
```

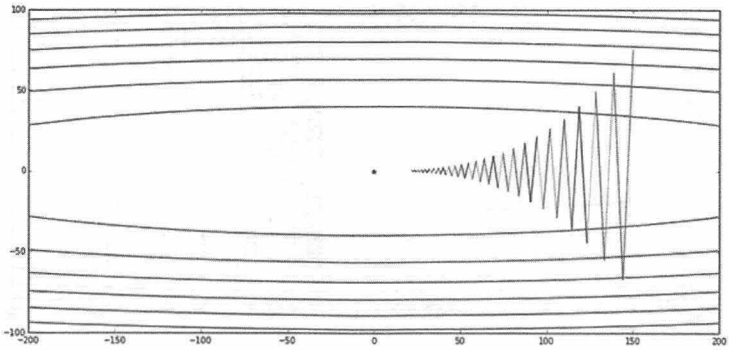


图 3-4 增大步长的梯度下降法优化曲线

虽然优化效果有了一定的提高，但成效依然不明显，而且优化的过程图中出现了参数值左右抖动的现象，这是怎么回事呢？看上去参数在优化过程中产生了某种“打转儿”的现象，做了很多无用功。看到这个曲线路径，读者可能会想到一个比较熟悉的极限运动，如图 3-5 所示。

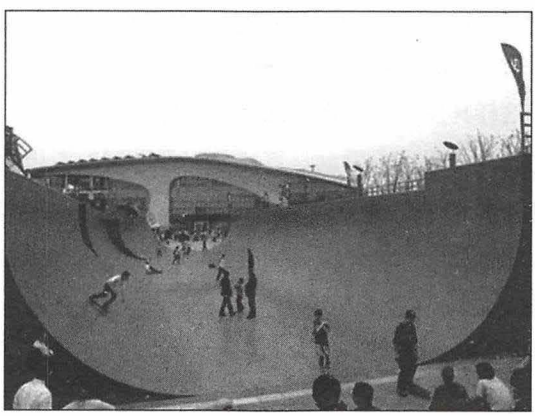


图 3-5 极限运动中的 U 形赛道

实际上，算法眼中优化曲面和这张图很像，算法没有让大家“失望”，一直在选择梯度下降最快的方向前进，但梯度下降最快的方向不一定能为优化提供太多的帮助，因此沿着这条道路进行优化就十分艰难——就像图中做极限运动的滑板少年一样，从一边的高台滑下，然后滑到另一边，不断反复。这个过程很像“打转”现象，朝正确优化方法前进的步伐比较小，而无效重复的前进步伐比较大。虽然优化的效率很低，但这就是梯度下降法。它的眼中只有梯度，并且只相信梯度。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

如果继续增加步长，优化曲线会变成什么样子呢？“滑板少年”还能不能再快点前进呢？实验结果如图 3-6 所示。

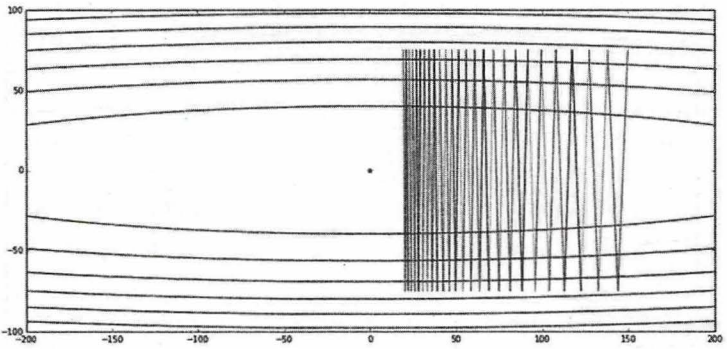


图 3-6　极限步长下梯度下降法的优化曲线

```
res, x_arr = gd([150,75], 0.02, g)
contour(X,Y,Z, x_arr)
```

从结果来看，这是滑板少年能尽的最大力了 …… 这个步长已经是能设置的最大步长，如果步长再设大些，滑板少年就要从 U 形赛道飞出去，优化参数的梯度也将发散出去。在这个问题中，由于两个坐标轴方向的函数的陡峭性质不同，两个方向对最大步长的限制不同，显然 y 方向对步长的限制更严格。但满足 y 方向的更新，x 方向就无法获得充分的更新，这样梯度下降法将无法获得很好的效果。

下面我们将目光转向动量，看看动量算法如何帮助这位滑板少年。

在开始动量算法计算前，首先要对滑板少年的行动方向做一个总结，读者会发现滑板少年每一次的行动只会在以下三个方向进行。

- 沿 $-x$ 方向滑行。
- 沿 $+y$ 方向滑行。
- 沿 $-y$ 方向滑行。

这样看来，要是少年能把行动的力量集中在往 $-x$ 方向走而不是沿 y 方向打转就好了。

这个想法可以被动量算法实现。我们可以想象，使用了动量后，历史的更新量会以衰减的形式不断作用在这些方向上，那么沿 $-y$ 和 $+y$ 两个方向的动量就可以相互抵消，而 $-x$ 方向的力则会一直加强，这样滑板少年虽然还会沿 y 方向打转，但是他在 $-x$ 方向的速度会因为之前的累积变得越来越快。

基于上面的分析，下面来看看加了动量技能的滑板少年的优化表现，如图 3-7 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

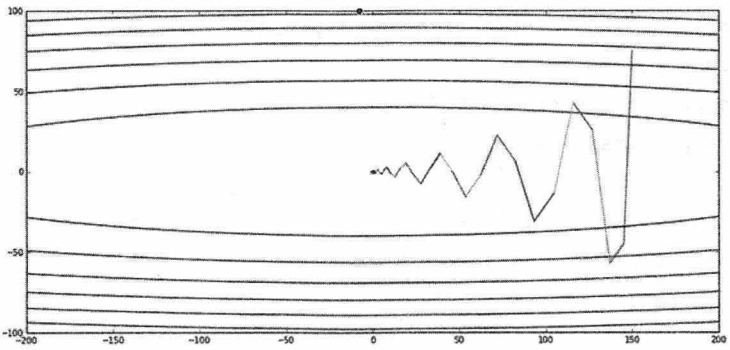


图 3-7 动量算法的优化曲线

```
momentum([150,75], 0.016, g)
```

优化曲线果然没有令人失望，尽管滑板少年还是有打转现象，但是在 50 轮迭代后，他进入了最优点的邻域，完成了优化任务，和前面的梯度下降法相比有了很大的进步。

当然，还是暴露了动量优化存在的一点问题，前面几轮迭代过程中目标值在 y 轴上的震荡比过去还要大。在发明动量算法后，又有科研人员发明了基于动量算法的改进算法，解决了动量算法没有解决的问题——更强烈的抖动，干脆让滑板少年停止玩耍，专心赶路。这就是 **Nesterov 算法** [2]。这里给出代码和刚才问题的优化过程，如图 3-8 所示。

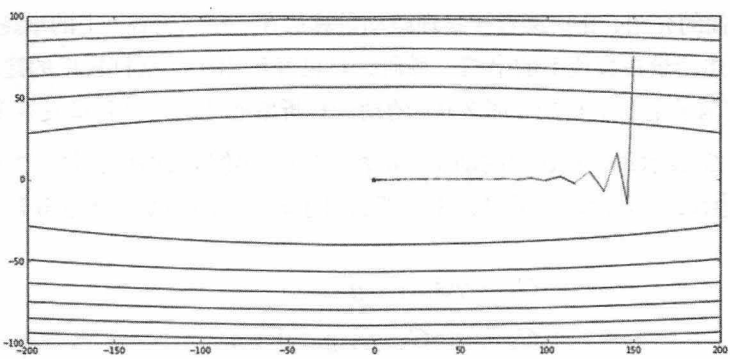


图 3-8 Nesterov 算法的优化曲线

```
def nesterov(x_start, step, g, discount = 0.7):
    x = np.array(x_start, dtype='float64')
    pre_grad = np.zeros_like(x)
    for i in range(50):
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
        x_future = x - step * discount * pre_grad
        grad = g(x_future)
        pre_grad = pre_grad * 0.7 + grad
        x -= pre_grad * step

        print '[ Epoch {0} ] grad = {1}, x = {2}'.format(i, grad, x)
        if abs(sum(grad)) < 1e-6:
            break;
    return x

nesterov([150,75], 0.012, g)
```

滑板少年不再贪玩，放弃在 U 形赛道上摩擦。那么，Nesterov 算法和动量算法相比有什么区别呢？动量算法计算的梯度是在当前的目标点的；而 Nesterov 算法计算的梯度是在动量更新后的优化点的。这其中关键的区别在于计算梯度的点。

读者可以想象一个场景，当优化点已经积累了某个抖动方向的梯度后，对动量算法来说，虽然当前点的梯度指向积累梯度的相反方向，但是量不够大，所以最终的优化方向还会在积累的方向上前进一段，这就是图 3-8 所示的效果。对 Nesterov 方法来说，如果按照积累方法再向前多走一段，则梯度中指向积累梯度相反方向的量会变大许多，最终两个方向的梯度抵消，反而使抖动方向的量迅速减少。Nesterov 的衰减速度确实比动量方法的快不少。

经过上面的介绍，相信读者能够理解动量算法在“穿越山谷”上的卓越表现。本节的最后来讲述动量在数值上的事情。科研人员已经给出了动量打折率的建议配置——0.9（刚才的例子全部是 0.7），那么 0.9 的动量打折率能使历史更新量发挥多大作用呢？

如果用 $\boldsymbol{G}$ 表示每一轮迭代的动量，$\boldsymbol{g}$ 表示当前一轮迭代的更新量（方向 × 步长），t 表示迭代轮数，γ 表示动量的打折率，那么对于时刻 t 的梯度更新量就有公式

$$
\begin{aligned}
\boldsymbol{G}_t &= \gamma \boldsymbol{G}_{t-1} + \boldsymbol{g}_t \\
\boldsymbol{G}_t &= \gamma(\gamma \boldsymbol{G}_{t-2} + \boldsymbol{g}_{t-1}) + \boldsymbol{g}_t \\
\boldsymbol{G}_t &= \gamma^2 \boldsymbol{G}_{t-2} + \gamma \boldsymbol{g}_{t-1} + \boldsymbol{g}_t \\
\boldsymbol{G}_t &= \gamma^t \boldsymbol{g}_0 + \gamma^{t-1} \boldsymbol{g}_1 + \cdots + \boldsymbol{g}_t
\end{aligned}
$$

这样可以计算出对于第一轮迭代的更新 $\boldsymbol{g}_0$ 来说，从 $\boldsymbol{G}_0$ 到 $\boldsymbol{G}_T$，它的总贡献量为

$$
(\gamma^t + \gamma^{t-1} + \cdots + \gamma + 1)\boldsymbol{g}_0
$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

相信读者已经发现它的贡献和是一个等比数列的和，比值为 γ。如果 $\gamma = 0.9$，那么根据公比小于 1 的等比数列的极限公式，可以知道更新量在极限状态下的贡献值

$$\lim_{t\to\infty}(\gamma^t + \gamma^{t-1} + \cdots + \gamma + 1)\boldsymbol{g}_0 = \frac{\boldsymbol{g}_0}{1-\gamma}$$

那么当 $\gamma = 0.9$ 时，它一共贡献了相当于自身 10 倍的能量。如果 $\gamma = 0.99$ 呢？那就是 100 倍能量。在实际应用中，打折率的设置需要分析具体任务中更新量需要多“持久”的动力。

3.3 共轭梯度法

本章的前面部分介绍了有关一阶梯度下降法的优化方法。本节介绍另一种一阶梯度方法：共轭梯度法。它也是活跃在优化世界的一个经典算法。它的计算速度还算不错，方法也相对简单，虽然比梯度下降法复杂不少，但是和二阶方法相比，也不算特别困难。

在前面的问题中，我们并没有给出待优化问题的具体形式，本节中为了使问题更具体，我们将待优化的问题形式定义为

$$\min_{\boldsymbol{X}} \frac{1}{2}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{X} - \boldsymbol{b}^{\mathrm{T}}\boldsymbol{X}$$

其中 $\boldsymbol{X}$ 为待优化的向量，$\boldsymbol{A}$ 为半正定的矩阵，$\boldsymbol{b}$ 为已知向量，对公式进行求导，并令导数为 0，就可以得到

$$\boldsymbol{A}\boldsymbol{X}^* = \boldsymbol{b}$$

其中 $\boldsymbol{X}^*$ 为最优值。从公式的形式来看，虽然可以通过高斯消元法直接解方程，但计算量比较大，计算复杂度比较高，所以我们希望通过优化的方式找到最优解。

3.3.1 精妙的约束

我们发现虽然梯度下降法的每一步都朝着局部最优的方向前进，但它在不同的迭代轮数会选择非常相近的方向，这说明当某一次选择了一个更新方向和步长后，这个方向并没有被更新完，未来还会存在这个方向的残差。如果把参数更新的轨迹显示出来，我们可以看到有时轨迹会走成 zig-zag 的形状，如 3.2 节的图 3-4、图 3-6 和图 3-7 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

于是，我们对优化提出了更高的要求——能不能在选择优化方向和步长上更智能，每朝一个方向走，就把这个方向走到“极致”？所谓极致，可以理解为在优化过程中再也不需要朝这个方向更新了。于是我们引出了一个变量——误差，假设最优的参数为 $\boldsymbol{x}^*$，当前第 t 轮的参数为 $\boldsymbol{x}_t$，误差可以定义为

$$\boldsymbol{e}_t = \boldsymbol{x}^* - \boldsymbol{x}_t$$

这个误差表示了参数的最优点和当前点之间的距离。那么目标就更明确了，我们希望每一步优化后，当前的误差和刚才优化的方向正交。请读者注意，这里我们没有使用梯度这个词，而是使用优化方向，因为从现在开始，我们的优化方向不一定是梯度。现在令 $\boldsymbol{r}_t$ 表示第 t 轮的更新方向，就可以得到下面的公式

$$\boldsymbol{r}_t^{\mathrm{T}} \boldsymbol{e}_{t+1} = 0$$

这真的是一个很漂亮的公式，假设每一轮的优化量都和误差正交，那么如果我们的优化空间有 d 维，理论上最多只需要迭代 d 轮就可以求解出来，这样在优化时间上就有了保证。如果我们直接使用这个公式，就会发现一个问题：公式中需要知道误差。换句话说，我们需要知道最优点在哪儿。如果我们知道最优点在哪儿，就不用优化了。可是不知道最优点儿，这个方法又无法直接使用，于是我们仿佛掉入了“先有鸡还是先有蛋”的陷阱。

不过庆幸的是，我们还有数学武器在手，可以用线性代数的知识偷天换日，解决这个问题，让这个美好的约束发挥它的作用。

3.3.2 共轭

为了解决 3.3.1 节提到的问题，我们要使用**共轭**这个工具。相信读者看到共轭这个词的时候会想，这词是什么意思？共轭中的轭实际上是一个工具，如图 3-9 所示。

轭就是绑在两头牛身上的木头，它让两个本来独立的个体变成了一个整体，属于牵线搭桥的关键道具。我们再回到刚才的问题，前面我们提到优化方向和误差正交，如果使用了共轭这个工具，现在两者的关系将变为共轭正交，也就是存在一个矩阵 $\boldsymbol{A}$（$\boldsymbol{A}$ 就是轭），使得优化方向和误差这两头牛“牵手成功”，满足正交的性质：

$$\boldsymbol{r}_t^{\mathrm{T}} \boldsymbol{A} \boldsymbol{e}_{t+1} = 0$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用



图 3-9 轭的示意图

看到这里可能有的读者依然很迷惑，为什么要加入这个矩阵，它的加入不就使原本正交的两项变得不正交了吗？别着急，其实如果我们在上面的原始公式中间加一个特殊的矩阵，例如单位阵，看上去就舒服很多了：

$$\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{I}\boldsymbol{e}_{t+1}=0$$

加入单位阵并不会改变结果，所以原本正交的二者依然正交。所以换个角度理解，我们以前简单的正交都是在单位阵这个简单且性质优良的“轭”的牵引下实现的。就好比图 3-9 中，两头老黄牛的“身高”十分接近，因此牵引它们的轭是一个水平的轭，现在来了一大一小两头黄牛，就需要一个不那么水平的轭把它们绑定在一起。经过轭的作用，原本不那么般配的一对牛也可以协同工作。所以回到正交中，如果使用单位阵作为轭，那么绑定在一起的就是常规意义上正交的一对向量；如果使用其他矩阵，那么共轭正交的向量肯定也会满足其他的性质。

从线性代数的角度解释，我们可以认为这个矩阵要完成线性变换的作用，将一个向量从一个线性空间转换到另一个空间。在第 2 章我们已经介绍了矩阵的作用，矩阵和向量相乘可以想象成是矩阵列向量的加权求和。在上面的公式中，我们通过计算 $\boldsymbol{A}\boldsymbol{e}_{t+1}$ 将误差向量映射到矩阵 $\boldsymbol{A}$ 组成的列空间中，只要优化方向与这个新的向量正交，就可以满足上面的例子。在优化过程中，如果 $\boldsymbol{A}$ 固定不变，就可以认为它们只是从一个空间的正交关系变成另一个空间的正交关系，其他的并没有什么不同。

那么有没有更实际的例子呢？例如我们有两个向量：

$$\boldsymbol{a}=[1,1],\boldsymbol{b}=[1,0.5]^{\mathrm{T}}$$

很显然它们不是正交的，但如果多了一个矩阵 $\boldsymbol{A}$：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$\boldsymbol{A} = [[1, 0][0, -2]]$$

就会发现，$\boldsymbol{b}$ 经过 $\boldsymbol{A}$ 转换后就与 $\boldsymbol{a}$ 正交了。

如果将上面两个向量的计算过程在坐标图上画出来，就可以得到这样的过程，首先如图 3-10 和图 3-11 所示，两幅图分别展示了在单位阵 $\boldsymbol{I}$ 和矩阵 $\boldsymbol{A}$ 共轭作用下的坐标轴形式。图 3-10 展示了以单位阵为轭的坐标形式，大家都比较熟悉，就是我们常见的坐标轴形式。图 3-11 展示了以矩阵 $\boldsymbol{A}$ 为轭的坐标形式，我们可以看出它在 x 轴没有发生改变，但是在 y 轴数值扩大一倍，并指向了相反方向。我们分别在两个坐标系下观察这两个向量，发现它们都不会产生正交。

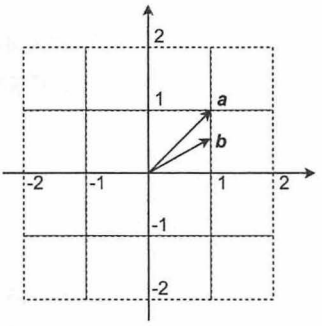


图 3-10　单位阵为轭的坐标图

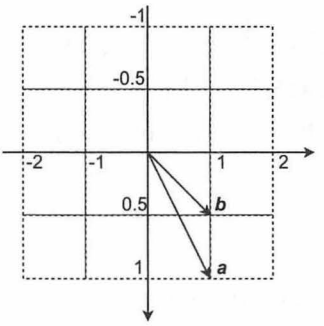


图 3-11　矩阵 $\boldsymbol{A}$ 为轭的坐标图

如图 3-12 所示，如果我们把单位阵坐标系中的 $\boldsymbol{a}$ 和矩阵 $\boldsymbol{A}$ 坐标系中的 $\boldsymbol{b}$ 放在一起，会发现两者产生了正交。所以我们也可以理解为原本不是 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 正交，而是 $\boldsymbol{a}$ 和另外一个向量 $\boldsymbol{b}'$ 正交，但是 $\boldsymbol{b}'$ 经过从矩阵 $\boldsymbol{A}$ 的还原得到了 $\boldsymbol{b}$。

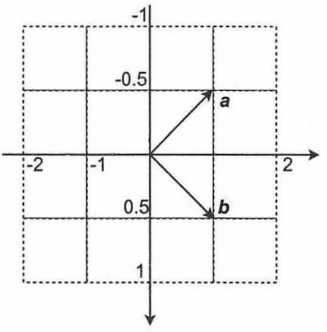


图 3-12　两个向量结合的效果

所以这个新的正交条件实际上只是多绕了一个弯，和前面的正交条件差距并不大。一旦接受了这样的设定，下面的内容就很好理解了。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.3.3 优化步长的确定

明确了共轭梯度法的目标和特点，我们就要开始推导算法公式了。共轭梯度法属于线搜索的一种，因此和梯度下降法类似，我们的总体思路不变，优化过程分如下两步：

（1）确定优化方向。

（2）确定优化步长。

在共轭梯度法中，确定优化方向比确定优化步长麻烦，因此我们先来介绍第二步（优化步长）的计算方法。假设当前的参数为 $\boldsymbol{X}_t$，我们已经得到了优化方向 $\boldsymbol{r}_t$，下面要确定的就是步长 α_t，根据前面提过的共轭正交公式

$$\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_{t+1}=0$$

我们可以开始推导：

$$\begin{aligned}\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_{t+1}&=\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}[\boldsymbol{e}_t+\boldsymbol{X}_t-\boldsymbol{X}_{t+1}]\\&=\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}[\boldsymbol{e}_t+\alpha_t\boldsymbol{r}_t]\\&=\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_t+\alpha_t\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_t=0\end{aligned}$$

于是可以整理得到

$$\alpha_t=-\frac{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_t}{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_t}$$

$$\alpha_t=-\frac{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}(\boldsymbol{X}^*-\boldsymbol{X}_t)}{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_t}$$

我们知道 $\boldsymbol{A}\boldsymbol{X}^*=\boldsymbol{b}$，第 t 轮的梯度 $\boldsymbol{g}_t=\boldsymbol{A}\boldsymbol{X}_t-\boldsymbol{b}$，于是公式最终变为

$$\alpha_t=\frac{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{g}_t}{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_t}$$

到这里，我们利用矩阵 $\boldsymbol{A}$ 成功地把公式中的误差 $\boldsymbol{e}$ 抵消，于是步长变得可解。

完成了步长的求解，接下来就要回到第一步看看优化方向的计算方法。我们要解决的主要问题是如何让优化方向和误差正交。由于每一次的优化后，剩下的误差和本次的优化正交（共轭正交），所以可以看出每一个优化方向彼此间都是正交的。那么，如何构建这些彼此正交的方向呢？

仅供非商业用途或交流学习使用

3.3.4 Gram-Schmidt 方法

在线性代数课上，我们曾经学过一个向量正交化的方法——Gram-Schmidt 方法。这个算法的输入是 N 维空间中 N 个线性无关的向量，由于向量间线性无关，任何一个向量都无法通过其他向量表达出来。算法的输出是 N 个相互正交的向量，也就是我们最终想要的向量组合。

它的具体算法如下，令输入向量为 $\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_N$，输出向量为 $\boldsymbol{d}_1, \boldsymbol{d}_2, \cdots, \boldsymbol{d}_N$，那么有：

（1）对于第一个向量，我们保持它不变：$\boldsymbol{u}_1 = \boldsymbol{d}_1$。

（2）对于第二个向量，我们去掉其中和第一个向量共线的部分，令去掉的比例为 β_i，所以第二个向量 $\boldsymbol{d}_2$ 等于 $\boldsymbol{u}_2 + \beta_1 \boldsymbol{d}_1$。

（3）对于第三个向量，我们去掉其中和第一、第二个向量共线的部分：$\boldsymbol{d}_3 = \boldsymbol{u}_3 + \sum_{i=1}^{2} \beta_{3,i} \boldsymbol{d}_i$。

（4）对于第 N 个向量，我们去掉其中和第一、第二、第 $N-1$ 个向量共线的部分：$\boldsymbol{d}_N = \boldsymbol{u}_N + \sum_{i=1}^{N-1} \beta_{N,i} \boldsymbol{d}_i$。

那么我们怎么求这些比例项 β 呢？我们利用前面提到的性质，向量之间正交（这里还是共轭正交），于是有

$$\boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_t = 0, (l = 1, 2, \ldots, t-1)$$

进一步展开，可以得到

$$\boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_t = \boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} (\boldsymbol{u}_t + \sum_{i=1}^{t-1} \beta_{t,i} \boldsymbol{d}_i) = \boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \boldsymbol{u}_t + \boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \sum_{i=1}^{t-1} \beta_{t,i} \boldsymbol{d}_i$$

利用正交的性质，可以将公式化简为

$$\boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} u_t + \boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \beta_{t,l} \boldsymbol{d}_l = 0$$

所以最终我们得到

$$\beta_{t,l} = -\frac{\boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \boldsymbol{u}_l}{\boldsymbol{d}_l^{\mathrm{T}} \boldsymbol{A} \boldsymbol{d}_l}$$

求得了这个变量，就可以依此求出当前向量与其他向量的共线部分，从而确保所有向量相互正交的性质。当然，我们也看到，为了实现正交的效果，算法的复杂度为 $O(N^2)$，在实现速度上不是很快。

仅供非商业用途或交流学习使用

3.3.5 共轭梯度

虽然 Gram-Schmidt 方法并不快，但它的确是一个可行的方案。到这里，我们还有如下两个问题没有解决。

（1）我们要用什么向量构建这些正交向量？

（2）随着向量数量的增加，我们需要计算的参数越来越多，我们一共要计算 $O(N^2)$ 数量级的 β，这个计算的数目还是有点多，能不能再减少呢？

上面的问题当然能够解决，方法也比较简单——用每一步的梯度构建这些正交向量，同时利用梯度的特点减少计算量。这就是算法名字的来源——共轭思想 + 梯度计算。

那么如何用梯度解决第二个问题呢？我们要依次证明下面三个推论。

（1）第 t 步计算的梯度 $\boldsymbol{g}_t$ 和前 $t-1$ 步的更新 $\{\boldsymbol{d}_j\}_{j=1}^{t-1}$ 正交。

（2）第 t 步计算的梯度 $\boldsymbol{g}_t$ 和前 $t-1$ 步的梯度 $\{\boldsymbol{g}_t\}_{j=1}^{t-1}$ 正交。

（3）第 t 步计算的梯度 $\boldsymbol{g}_t$ 和前 $t-2$ 步的更新 $\{\boldsymbol{d}_j\}_{j=1}^{t-2}$ 共轭正交。

先来证明第一个推论。前面提到共轭梯度法的基本思想：每一轮把某一方向优化彻底，保证后面的优化不再对这个方向做任何操作。假设算法一共进行了 T 轮迭代，我们就可以用这 T 轮求出的优化方向组合成最终的误差。

假设我们的初始参数值为 $\boldsymbol{x}_1$，那么它到最优点 $\boldsymbol{x}^*$ 的距离为 $\boldsymbol{e}_1$，这也是当前的误差。第 t 步求出的优化方向为 $\boldsymbol{d}_t$，每一轮的优化步长为 γ_i，根据上面的定义，可以得到如下公式

$$\boldsymbol{e}_1 = \sum_{i=1}^{T} \gamma_i \boldsymbol{d}_i$$

这样我们就用更新方向表示了误差。对于不同迭代轮数的误差，我们也可以用类似的公式表示。上面的公式利用所有的更新组合成误差，我们反过来思考也可以理解，通过每一步的更新，算法最终收敛，误差变为 0。所以，这里的 γ 和前面提到的步长 α 之间是等价的关系。

下面就来证明：

当 $i < j$ 时，

$$\boldsymbol{d}_i^{\mathrm{T}} \boldsymbol{g}_j = 0$$

证明过程如下

$$\boldsymbol{d}_i^{\mathrm{T}} \boldsymbol{g}_j = \boldsymbol{d}_i^{\mathrm{T}} (\boldsymbol{A}\boldsymbol{X}_j)$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= \boldsymbol{d}_i^{\mathrm{T}}(\boldsymbol{A}\boldsymbol{X}_j - \boldsymbol{A}\boldsymbol{X}^*) \\
&= \boldsymbol{d}_i^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_j \\
&= \boldsymbol{d}_i^{\mathrm{T}}\boldsymbol{A}(\sum_{t=j}^{T}\gamma_t\boldsymbol{d}_t)
\end{aligned}
$$

因为前面已经说明任意两个优化方向是相互正交的（Gram-Schmidt 方法），所以这个式子的结果为 0。利用共轭正交的性质，我们得到了更好的正交性质，这就是共轭梯度法最精髓的地方。

下面证明第二个推论。我们直接给出推导过程。

$$
\boldsymbol{g}_i^{\mathrm{T}}\boldsymbol{g}_j = (\boldsymbol{d}_i + \sum_{j=1}^{i-1}\beta_j\boldsymbol{d}_j)^{\mathrm{T}}\boldsymbol{g}_j = 0
$$

此时我们发现，由于每一步的梯度与此前的梯度正交，因此我们可以使用它作为线性无关的向量组，从而组成共轭正交的向量组。

第三个推论将以第二个推论为基础，也就是

$$
\boldsymbol{g}_i^{\mathrm{T}}\boldsymbol{g}_j = 0(i < j)
$$

将公式进行变换，可以得到

$$
\begin{aligned}
\boldsymbol{g}_{j+1}^{\mathrm{T}}\boldsymbol{g}_i &= (\boldsymbol{A}\boldsymbol{X}_{j+1})^{\mathrm{T}}\boldsymbol{g}_i \\
&= (\boldsymbol{A}(\boldsymbol{X}_j + \alpha_j\boldsymbol{d}_j))^{\mathrm{T}}\boldsymbol{g}_i \\
&= (\boldsymbol{A}\boldsymbol{X}_j)^{\mathrm{T}}\boldsymbol{g}_i + (\boldsymbol{A}\alpha_j\boldsymbol{d}_j)^{\mathrm{T}}\boldsymbol{g}_i \\
&= \boldsymbol{g}_j^{\mathrm{T}}\boldsymbol{g}_i + \alpha_j\boldsymbol{d}_j^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{g}_i \quad （\boldsymbol{A} 是对称矩阵） \\
&= \boldsymbol{g}_j^{\mathrm{T}}\boldsymbol{g}_i + \alpha_j\boldsymbol{d}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{g}_i
\end{aligned}
$$

对公式进行变换，可以得到

$$
\boldsymbol{d}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{g}_i = \frac{1}{\alpha_j}[\boldsymbol{g}_{j+1}^{\mathrm{T}}\boldsymbol{g}_i - \boldsymbol{g}_j^{\mathrm{T}}\boldsymbol{g}_i]
$$

此时我们发现公式的左边就是求解 β 的分子，那么公式右边等于什么呢？我们令 $j < i$，于是有：

- 当 $j = i - 1$ 时，等式右边为 $\frac{1}{\alpha_j}[\boldsymbol{g}_i^{\mathrm{T}}\boldsymbol{g}_i - \boldsymbol{g}_{i-1}^{\mathrm{T}}\boldsymbol{g}_i]$，可以看出当 $\boldsymbol{g}_i$ 不等于 0 时，这

仅供非商业用途或交流学习使用

部分公式不等于 0。

- 当 $j < i-1$ 时，等式右边为 $\frac{1}{\alpha_j}[\boldsymbol{g}_{j+1}^{\mathrm{T}}\boldsymbol{g}_i - \boldsymbol{g}_j^{\mathrm{T}}\boldsymbol{g}_i]$，由于 $j+1 < i$，$j < i$，根据前面的推断，这部分公式为 0。

所以可以得到，使用 Gram-Schmidt 方法可以保证更新方向共轭正交。对于第 t 轮优化，我们只计算 β_{t-1} 即可，其他的 β 值均为 0，不用计算，这样计算量得到了极大的降低。

到这里我们完成了共轭梯度法的理论推导，由于算法比较复杂，我们需要对算法进行回顾。

（1）最初，我们希望我们的更新方向与误差正交，从而得出了 $\boldsymbol{d}_i^{\mathrm{T}}\boldsymbol{e}_i = 0$。

（2）由于误差无法提前知道，直接计算正交比较困难，所以我们将正交改为共轭正交，于是公式变为 $\boldsymbol{d}_i^{\mathrm{T}}\boldsymbol{A}\boldsymbol{e}_i = 0$。

（3）我们希望使用函数的梯度作为优化方向的基础，每一次求出当前的梯度，就用梯度减去之前更新方向的成分，以保证每一次的优化都保持共轭正交。

（4）我们采用 Gram-Schmidt 方法确保更新方向能够维持共轭正交的性质，根据证明，我们发现在使用梯度时，每一轮只需要计算一个 β 值就能实现共轭正交的效果，计算量大大降低。β 的求解方法为 $\beta_{t,l} = -\frac{\boldsymbol{d}_l^{\mathrm{T}}\boldsymbol{A}\boldsymbol{u}_l}{\boldsymbol{d}_l^{\mathrm{T}}\boldsymbol{A}\boldsymbol{d}_l}$。

（5）完成了更新方向的计算，我们根据前面的推论，计算每一步的步长：$\alpha_t = \frac{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{g}_t}{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_t}$。

至此不免有些感慨，向前辈们的想象力致敬，能够利用一些看上去抽象又无关紧要的定理帮助我们解决这些大问题，这里面存在的不仅是数学的美丽，甚至是哲学的精彩……

下面我们来简单介绍这个算法的实现，在第 5 章将介绍的 Baselines 项目中，baselines/common/cg.py 文件中给出了共轭梯度法的实现，代码如下所示。

```
import numpy as np
def cg(f_Ax, b, cg_iters=10, callback=None, verbose=False, residual_tol=1e
-10):
    p = b.copy()
    r = b.copy()
    x = np.zeros_like(b)
    rdotr = r.dot(r)
    for i in range(cg_iters):
        z = f_Ax(p)
        v = rdotr / p.dot(z)
```

仅供非商业用途或交流学习使用


```
        x += v*p
        r -= v*z
        newrdotr = r.dot(r)
        mu = newrdotr/rdotr
        p = r + mu*p
        rdotr = newrdotr
        if rdotr < residual_tol:
            break
    return x
```

乍一看这份代码，相信读者也有些疑惑，看上去代码的实现和前面的理论并不相同，这其中有什么区别呢？

我们的目标是求出 $\boldsymbol{AX}=\boldsymbol{b}$ 的解。代码中的残差为 $\boldsymbol{r}$，真正的优化方向为 $\boldsymbol{p}$，优化步长为 v。算法默认的 $\boldsymbol{X}$ 初始值为 0，所以初始的残差 $\boldsymbol{r}_1=\boldsymbol{b}$，由于第 1 轮我们没有一个已经执行的优化方向，因此优化方向 $\boldsymbol{p}_1=\boldsymbol{r}_1$，代码的第 3~5 行实现了上述功能。由于 $\boldsymbol{r}^{\mathrm{T}}\boldsymbol{r}$ 将在后面反复使用，于是在代码第 6 行我们计算了 rdotr 的值。

在后面的循环中，我们首先要计算步长，前面给出的计算步长的公式为 $\alpha_t=\frac{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{r}_t}{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_t}$，对其进行变换，可以得到

$$\begin{aligned}\alpha_t&=\frac{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{r}_t}{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_t}\\&=\frac{(\boldsymbol{r}_t-\beta_{t-1}\boldsymbol{p}_{t-1})^{\mathrm{T}}\boldsymbol{r}_t}{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_t}\\&=\frac{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{r}_t}{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_t}\end{aligned}$$

于是第 8 行和第 9 行代码实现了计算 $\boldsymbol{A}\boldsymbol{p}_t$ 和 α_t 的功能。其中第 8 行的函数由用户提供，用于实现 $\boldsymbol{A}\boldsymbol{p}_t$。接下来就是更新 $\boldsymbol{X}$，于是第 10 行实现了对 $\boldsymbol{X}$ 的更新操作：$\boldsymbol{X}_{t+1}=\boldsymbol{X}_t+\alpha_t\boldsymbol{p}_t$。

得到了新的 $\boldsymbol{X}$，我们就可以计算下一轮的梯度，也就是残差 $\boldsymbol{r}_{t+1}$。正常计算残差的方法为 $\boldsymbol{r}_{t+1}=\boldsymbol{b}-\boldsymbol{A}\boldsymbol{X}_{t+1}$，公式经过变换，就得到了代码中第 11 行的公式

$$\begin{aligned}\boldsymbol{r}_{t+1}&=\boldsymbol{b}-\boldsymbol{A}\boldsymbol{X}_{t+1}\\&=\boldsymbol{b}-\boldsymbol{A}(\boldsymbol{X}_t+\alpha_t\boldsymbol{p}_t)\\&=\boldsymbol{b}-\boldsymbol{A}\boldsymbol{X}_t-\alpha_t\boldsymbol{A}\boldsymbol{p}_t\\&=\boldsymbol{r}_t-\alpha_t\boldsymbol{A}\boldsymbol{p}_t\end{aligned}$$

仅供非商业用途或交流学习使用

得到残差后，下一步就是计算真正的优化方向，前面推导的公式为 $\boldsymbol{p}_{t+1} = \boldsymbol{r}_{t+1} - \frac{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_{t+1}}{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_t}\boldsymbol{p}_t$，经过变换，我们就能得到代码中第 12~14 行的公式

$$
\begin{aligned}
\boldsymbol{p}_{t+1} &= \boldsymbol{r}_{t+1} - \frac{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{r}_{t+1}}{\boldsymbol{p}_t^{\mathrm{T}}\boldsymbol{A}\boldsymbol{p}_t}\boldsymbol{p}_t \\
&= \boldsymbol{r}_{t+1} - \frac{(\boldsymbol{A}\boldsymbol{p}_t)^{\mathrm{T}}\boldsymbol{r}_{t+1}}{(\boldsymbol{A}\boldsymbol{p}_t)^{\mathrm{T}}\boldsymbol{p}_t}\boldsymbol{p}_t \\
&= \boldsymbol{r}_{t+1} - \frac{(\frac{\boldsymbol{r}_t-\boldsymbol{r}_{t+1}}{\alpha})^{\mathrm{T}}\boldsymbol{r}_{t+1}}{(\frac{\boldsymbol{r}_t-\boldsymbol{r}_{t+1}}{\alpha})^{\mathrm{T}}\boldsymbol{p}_t}\boldsymbol{p}_t \\
&= \boldsymbol{r}_{t+1} - \frac{(\boldsymbol{r}_t-\boldsymbol{r}_{t+1})^{\mathrm{T}}\boldsymbol{r}_{t+1}}{(\boldsymbol{r}_t-\boldsymbol{r}_{t+1})^{\mathrm{T}}(\boldsymbol{r}_t-\beta_{t-1}\boldsymbol{p}_{t-1})}\boldsymbol{p}_t
\end{aligned}
$$

根据残差和更新方向的正交关系，可以得到

$$
= \boldsymbol{r}_{t+1} + \frac{\boldsymbol{r}_{t+1}^{\mathrm{T}}\boldsymbol{r}_{t+1}}{\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{r}_t}\boldsymbol{p}_t
$$

这样就和代码一致了，我们可以复用 $\boldsymbol{r}_t^{\mathrm{T}}\boldsymbol{r}_t$ 这样的中间计算结果。

第 15~17 行代码用于检查当前的梯度值，当梯度小于一定值时，我们可以认为算法达到了一定的精度，于是停止优化，算法结束；如果梯度仍比较大，就继续进行下一轮优化，此时我们已经知道了 $\boldsymbol{r}_{t+1}$ 和 $\boldsymbol{p}_{t+1}$，所以接下来就是计算 α_{t+1}，也就是从第 8 行开始的又一次循环。

以上就是共轭梯度法的介绍，实际上共轭梯度法不止这些内容，而其他复杂的知识与本书的内容关系不大，这里不再赘述。第 10 章将介绍置信区间策略优化算法，那时还会涉及共轭梯度法的知识。

3.4 自然梯度法

3.4.1 基本概念

梯度下降法是一种十分直观的优化方法，它的计算比较简单，实现也比较简单，但是它存在一些问题。我们从 3.2 节的例子中看到，当优化问题的两个坐标轴的尺度差异比较大时，使用一个统一的学习率会产生问题：某一个坐标轴可能会发散。那么产生这个问题的根源是什么呢？

客观地说，优化问题中参数尺度不同是很难避免的。因此，在不同的优化曲面上，虽然每一轮迭代对参数的更新量相差不多，但是它们对模型的影响完全不同，有时会

仅供非商业用途或交流学习使用

使模型有较大的改进，有时对模型产生的改进较小，有时甚至会使模型倒退，这些效果都可以在 loss 曲线上体现出来。

这样我们就发现了一个很关键的问题：梯度下降法针对的目标是参数，我们更新的目标也是参数，而优化的真正目标是找到最优的模型。在每一轮迭代中，对参数的改进和对模型的改进是不同的。对不同参数进行同样数值大小的更新，不一定带来同样的模型改变，反过来也是如此。那么有没有一种方法，不简单地使用学习率对参数更新进行量化，而是对模型效果进行量化呢？这就是自然梯度下降法（Natural Gradient Descent）的思想来源。

一般来说，我们会假设模型的参数来自高维的实数空间 $\mathbf{R}^d$，在这样的空间中，欧氏距离是空间的度量方法，即对空间内的两个点 x 和 y，它们的距离（d）为：

$$d(x,y) = \|x-y\|_2^2 = \sum_i (x_i - y_i)^2$$

如果我们的模型有 N 个参数，那么模型的所有参数可以组成一个长度为 N 的向量。我们假设存在两组参数 θ_1 和 θ_2，对应的模型为 $f(x;\theta_1)$ 和 $f(x;\theta_2)$，两组参数的距离可以通过欧式距离体现，那么这两个模型的距离（也就是表现效果）该如何体现呢？

在 2.6 节我们曾介绍过 KL 散度，它可以比较两个概率分布之间的差异。如果我们认为模型最终的输出是一个概率分布，就可以使用它来表示：

$$\mathrm{KL}(f(x;\theta_1), f(x;\theta_2)) = \int_x f(x;\theta_1) \log \frac{f(x;\theta_1)}{f(x;\theta_2)} \mathrm{d}x$$

很显然，参数和模型对距离的衡量方法是不同的。梯度下降法只考虑了在梯度方向上对参数进行更新，并没有考虑模型层面的更新，因此就会出现模型更新不均匀的现象。

对一些简单的优化问题来说，梯度下降法已经可以做得很好，但对一些复杂的问题来说，梯度下降法可能会难以优化，甚至造成发散，所以找到一个稳定优化的方法显得十分重要。

按照梯度下降法的思想，我们定义每一轮的迭代优化都要解决这样一个子问题：

$$\begin{aligned} &\mathrm{minimize}_{\Delta w} f(w+\Delta w) \\ &\text{s.t. } |\Delta w| < \epsilon \end{aligned}$$

将原函数进行一阶泰勒展开，问题就变为

仅供非商业用途或交流学习使用

$$\begin{aligned}&\text{minimize}_{\Delta w} f(w) + \nabla_w f(w)\Delta w \\ &\text{s.t. } |\Delta w| < \epsilon\end{aligned}$$

对问题求导，并对更新量做一定的限制，就可以得到梯度下降法。如果我们改为对模型的距离进行约束，就可以得到另外一个形式的公式

$$\begin{aligned}&\text{minimize}_{\Delta w} f(w) + \nabla_w f(w)\Delta w \\ &\text{s.t. } \mathrm{KL}(f(w), f(w+\Delta w)) < \epsilon\end{aligned}$$

这就是自然梯度法的优化形式。可以看出，有了模型层面的约束，每一轮迭代无论参数发生多大的变化，模型的变化都会限制在一定的范围内，因此不论我们使用什么样的模型，这个约束都会起到相同的效果，因此这个约束是具有普适性的，在任何模型上都能发挥同样稳定的效果。

自然梯度法的优点已经显而易见，所以让我们来看看它的缺点：这个 KL 散度该怎么计算、怎么优化呢？看上去这并不是一个容易处理的式子。

3.4.2 Fisher 信息矩阵

以往的经验一次又一次地告诉我们，当我们遇到一个难以计算的复杂公式时，最好先思考能不能化简它。通过计算，KL 散度可以变成熵和交叉熵的运算，具体的推导过程为

$$\begin{aligned}&\mathrm{KL}(f(w)\|f(w+\Delta w)) \\ &= E_{f_w}[\log\frac{f(w)}{f(w+\Delta w)}] \\ &= E_{f_w}[\log f(w)] - E_{f_w}[\log f(w+\Delta w)]\end{aligned}$$

其中 E 表示为依 $f(w)$ 概率计算得到的期望。对等式右边的第二项进行二阶泰勒展开，可以得到

$$\begin{aligned}&\approx E_{f_w}[\log f(w)] - E_{f_w}[\log f(w) + \nabla_w \log f(w)\Delta w + \frac{1}{2}\Delta w^{\mathrm{T}}\nabla_w^2 \log f(w)\Delta w] \\ &= E_{f_w}[\log f(w)] - E_{f_w}[\log f(w)] - E_{f_w}[\nabla_w \log f(w)\Delta w] - E_{f_w}[\frac{1}{2}\Delta w^{\mathrm{T}}\nabla_w^2 \log f(w)\Delta w] \\ &= -E_{f_w}[\nabla_w \log f(w)\Delta w] - E_{f_w}[\frac{1}{2}\Delta w^{\mathrm{T}}\nabla_w^2 \log f(w)\Delta w]\end{aligned}$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= -\int_x f(w)\nabla_w \log f(w)\Delta w \mathrm{d}x - \int_x f(w)\frac{1}{2}\Delta w^{\mathrm{T}}\nabla_w^2 \log f(w)\Delta w \mathrm{d}x \\
&= -[\int_x f(w)\frac{1}{f(w)}\nabla_w f(w)\mathrm{d}x]\Delta w - \frac{1}{2}\Delta w^{\mathrm{T}}[\int_x f(w)\nabla_w^2 \log f(w)\mathrm{d}x]\Delta w \\
&= -[\int_x \nabla_w f(w)\mathrm{d}x]\Delta w - \frac{1}{2}\Delta w^{\mathrm{T}} E_{f_w}[\nabla_w^2 \log f(w)]\Delta w
\end{aligned}
$$

由于我们定义的函数 $f(w)$ 一般都是一个连续、可导、有界、性质优良的函数，所以这里第一项的积分和微分可以互换，同时我们将简写的 $f(w)$ 用完整的形式写出：$f(x;w)$，于是上式的第一项就变为

$$
\begin{aligned}
&= -[\nabla_w \int_x f(x;w)\mathrm{d}x]\Delta w \\
&= -[\nabla_w 1]\Delta w \\
&= 0
\end{aligned}
$$

最终得到

$$
\mathrm{KL}(f(w)\|f(w+\Delta w)) = -\frac{1}{2}\Delta w^{\mathrm{T}} E_{f_w}[\nabla_w^2 \log f(w)]\Delta w
$$

这里包含一个二阶导的期望值，虽然看上去比 KL 散度直观，但它仍然比较复杂。我们需要用 Fisher 信息矩阵（Fisher Information Matrix）来表示它。Fisher 信息是信息几何中的一个概念，它也被应用到机器学习中。前面提到 $f(w)$ 表示某个概率分布，我们首先定义 Score 函数（Score Function）为对数似然函数的一阶导数

$$
l_{f_w} = \nabla_w \log f(x;w)
$$

通过计算可以发现，score 函数的期望值为 0，公式推导如下：

$$
\begin{aligned}
E_{f(w)}[l_{f_w}] &= \int_x f(w)\nabla_w \log f(w)\mathrm{d}x \\
&= \int_x f(w)\frac{\nabla_w f(w)}{f(w)}\mathrm{d}x \\
&= \int \nabla_w f(w)\mathrm{d}x \\
&= \nabla_w \int_x f(x;w)\mathrm{d}x \\
&= \nabla_w 1 \\
&= 0
\end{aligned}
$$

仅供非商业用途或交流学习使用

Fisher 信息矩阵可以通过 score 函数定义：

$$\begin{aligned}\boldsymbol{I}_{f_w} &= E_{f_w}[\nabla_w \log f(w) \nabla_w \log f(w)^{\mathrm{T}}] \\ &= E_{f_w}[\boldsymbol{l}_{f_w} \boldsymbol{l}_{f_w}^{\mathrm{T}}]\end{aligned}$$

Fisher 信息矩阵有什么用处呢？接下来我们要证明一个重要的结论：Fisher 信息矩阵和 KL 散度的负二阶导数相等。一旦这个定理证明成功，就可以用更简单的方法计算 KL 散度。首先将二阶导数进行变换：

$$\begin{aligned}\nabla_w^2 \log f(w) &= \nabla_w \left(\frac{\nabla_w f(w)}{f(w)}\right) \\ &= \frac{\nabla_w^2 f(w)}{f(w)} - \frac{\nabla f(w) \nabla_w f(w)^{\mathrm{T}}}{f(w)^2} \\ &= \frac{\nabla_w^2 f(w)}{f(w)} - \frac{\nabla_w f(w)}{f(w)} \frac{\nabla_w f(w)^{\mathrm{T}}}{f(w)} \\ &= \frac{\nabla_w^2 f(w)}{f(w)} - \nabla_w \log f(w) \nabla_w \log f(w)^{\mathrm{T}} \\ &= \frac{\nabla_w^2 f(w)}{f(w)} - \boldsymbol{l}_{f_w} \boldsymbol{l}_{f_w}^{\mathrm{T}}\end{aligned}$$

这样就得到了二阶导数和 score 函数的关系，进一步推导，可以得到

$$\begin{aligned}\boldsymbol{I}_{f_w} &= E_{f_w}[\boldsymbol{l}_{f_w} \boldsymbol{l}_{f_w}^{\mathrm{T}}] \\ &= E_{f_w}[-\nabla_w^2 \log f(w) + \frac{\nabla_w^2 f(w)}{f(w)}] \\ &= \int_x f(w)[-\nabla_w^2 \log f(w) + \frac{\nabla_w^2 f(w)}{f(w)}] \mathrm{d}x \\ &= -\int_x f(w) \nabla_w^2 \log f(w) \mathrm{d}x + \int_x \nabla_w^2 f(w) \mathrm{d}x\end{aligned}$$

我们假设表示概率分布的函数的二阶导存在并满足良好的性质，于是积分和求导符号可以进行互换，得到

$$\begin{aligned}&= -\int_x f(w) \nabla_w^2 \log f(w) \mathrm{d}x + \nabla_w^2 \int_w f(w) \mathrm{d}x \\ &= -\int_x f(w) \nabla_w^2 \log f(w) \mathrm{d}x + \nabla_w^2 1 \\ &= -\int_x f(w) \nabla_w^2 \log f(w) \mathrm{d}x\end{aligned}$$

仅供非商业用途或交流学习使用

$$= -E_{f_w}[\nabla_w^2 \log f(x)]$$

这样我们就得出在一定条件下（概率分布函数要具备良好的性质），Fisher 信息矩阵和 KL 散度二阶导的相反数相等。虽然这样的推导十分严谨，但对读者来说，这些抽象的概念很难让我们建立起直观的感觉。那么 Fisher 信息矩阵究竟是什么样的，它的近似效果如何呢？我们以最简单的伯努利分布为例，看看它的 Fisher 信息是什么。

伯努利分布可以写成

$$p(x;\theta) = \theta^x (1-\theta)^{1-x}, x \in \{0,1\}$$

那么 Fisher 信息矩阵为

$$\begin{aligned}
\boldsymbol{I}_p &= E_p[\boldsymbol{l}_p \boldsymbol{l}_p^{\mathrm{T}}] \\
&= \sum_{x=0}^{1} p(x;\theta) \nabla_\theta \log p(x;\theta) \nabla_\theta \log p(x;\theta)^{\mathrm{T}} \\
&= \sum_{x=0}^{1} p(x;\theta) \nabla_\theta [x\log\theta + (1-x)\log(1-\theta)] \nabla_\theta [x\log\theta + (1-x)\log(1-\theta)] \\
&= \sum_{x=0}^{1} p(x;\theta) [\frac{x}{\theta} - \frac{1-x}{1-\theta}][\frac{x}{\theta} - \frac{1-x}{1-\theta}]
\end{aligned}$$

展开公式，得到

$$\begin{aligned}
&= (1-\theta)[\frac{1}{1-\theta}][\frac{1}{1-\theta}] + \theta[\frac{1}{\theta}][\frac{1}{\theta}] \\
&= \frac{1}{1-\theta} + \frac{1}{\theta} \\
&= \frac{1}{(1-\theta)\theta}
\end{aligned}$$

利用 KL 散度的二阶导，同样可以得到

$$\begin{aligned}
&- E_p[\nabla_\theta^2 \log p(x;\theta)] = -\sum_{x=0}^{1} p(x;\theta) \nabla^2 \log p(x;\theta) \\
&= -\sum_{x=0}^{1} p(x;\theta) \nabla_\theta^2 [x\log\theta + (1-x)\log(1-\theta)] \\
&= -\sum_{x=0}^{1} p(x;\theta) \nabla_\theta [\frac{x}{\theta} - \frac{1-x}{1-\theta}]
\end{aligned}$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= -\sum_{x=0}^{1} p(x;\theta)[-\frac{x}{\theta^2} - \frac{1-x}{(1-\theta)^2}] \\
&= -(p(x=0;\theta)[-\frac{0}{\theta^2} - \frac{1-0}{(1-\theta)^2}] + p(x=1;\theta)[-\frac{1}{\theta^2} - \frac{1-1}{(1-\theta)^2}]) \\
&= -((1-\theta) * [-\frac{1}{(1-\theta)^2}] + \theta * [-\frac{1}{\theta^2}]) \\
&= \frac{1}{1-\theta} + \frac{1}{\theta} \\
&= \frac{1}{(1-\theta)\theta}
\end{aligned}
$$

因为它只有一个参数，因此这个 Fisher 信息矩阵只能算一个标量。我们假设存在一个分布为 $\theta = 0.5$ 的伯努利分布，希望求解其他伯努利分布和这个分布的距离。这里分别使用 KL 散度和 Fisher 信息矩阵近似两种方式计算另一个分布之间的距离。代码如下所示：

```
from math import log
import matplotlib.pyplot as plt
import numpy as np

def kl(p,q):
    return p * log(p / q) + (1 - p) * log((1 - p) / (1 - q))

def fisher(p):
    return 1 / p / (1 - p)

def fisher_kl(p,q):
    return 0.5 * fisher(p) * (p - q)*(p - q)

x = np.linspace(0.1, 0.9, 20)
y1 = [kl(0.5, q) for q in x]
y2 = [fisher_kl(0.5, q) for q in x]

plt.plot(x, y1, label='kl')
plt.plot(x, y2, label='fisher_kl')
plt.legend()
```

最终的结果如图 3-13 所示。

仅供非商业用途或交流学习使用

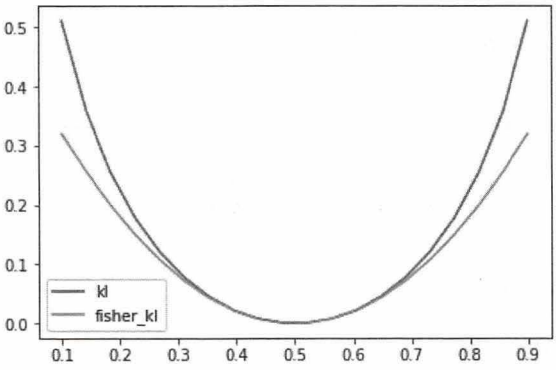


图 3-13　KL 散度与 Fisher 信息近似的对比图（$\theta = 0.5$ 的伯努利分布与另一个伯努利分布的比较）

由于我们选取的参照点为 $\theta = 0.5$ 的分布位置，可以看出在 0.5 附近，KL 散度和 Fisher 信息矩阵近似的 KL 散度十分接近。因此我们确信在局部范围内两者是可以相互替换的。这就是 Fisher 信息矩阵的实际意义。

3.4.3　自然梯度法目标公式

通过前面的推演，我们的目标函数变为

$$\begin{aligned}&\text{minimize}_{\Delta w} f(w) + \nabla_w f(w)\Delta w \\ &\text{s.t. } \frac{1}{2}\Delta w^{\mathrm{T}} \boldsymbol{I}_{f_w} \Delta w < \epsilon\end{aligned}$$

这个有约束的问题可以通过拉格朗日乘子法表示为

$$\text{minimize}_{\Delta w} f(w) + \nabla_w f(w)\Delta w + \lambda[\frac{1}{2}\Delta w^{\mathrm{T}} \boldsymbol{I}_{f_w} \Delta w - \epsilon]$$

对公式进行求导，并求解对应的极限点，可以得到

$$\nabla_w f(w) + \lambda \boldsymbol{I}_{f_w} \Delta w = 0$$

$$\Delta w = -\frac{1}{\lambda} \boldsymbol{I}_{f_w}^{-1} \nabla_w f(w)$$

公式中的 $\frac{1}{\lambda}$ 可以当作梯度下降法的学习率类似的分量，那么自然梯度法的优化方向就可以看作 $\boldsymbol{I}_{f_w}^{-1} \nabla_w f(w)$，与梯度下降法不同，它需要额外求解 Fisher 信息矩阵的逆。

仅供非商业用途或交流学习使用

熟悉优化算法的读者在看到这个公式形式后，一定会联想到牛顿法。牛顿法是一个二阶梯度算法，它求解优化方向的公式为

$$\boldsymbol{g} = -\nabla_w^2 f(w)^{-1} \nabla_w f(w)$$

我们很容易看出两个公式的区别与联系。根据 Fisher 信息矩阵的求解方法，自然梯度法可以变成一个一阶优化问题，也可以变成一个二阶优化问题。如果把它看作一个一阶优化问题，那么需要对优化步长做更多的考量；如果把它看作一个二阶优化问题，那么牛顿法中可能遇到的一些问题同样会在自然梯度法中出现。

实际上，自然梯度法是一个十分复杂的方法，本节只介绍基本原理，在第 10 章中，我们将继续介绍基于此方法进行优化的细节。

3.5 总结

本节我们主要介绍了 4 种优化算法，让我们在此做一个总结。

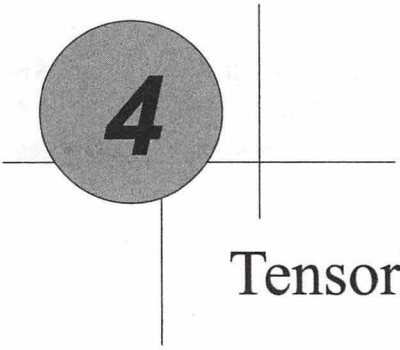
（1）梯度下降法：算法简单直接，但是在设计优化步长时需要认真考量。

（2）动量法：更好地穿越一些平坦的优化区域。

（3）共轭梯度法：强调每一步优化迭代的质量。

（4）自然梯度法：将每一轮迭代中对参数的更新转变为对模型效果的更新。

仅供非商业用途或交流学习使用



TensorFlow 入门

古语讲：“工欲善其事，必先利其器”，拥有好的工具才能完成更大的挑战。为了更快地实现机器学习相关的工作，我们需要有一个强大的工具帮我们管理数据、构建模型、完成训练与预测的工作。本章介绍一款优秀的工具——TensorFlow。TensorFlow 是一款由 Google 开发的使用数据流图进行数值计算的开源软件库 [1]。由于它具有使用便捷、功力强大、扩展性好等特点，现在已经被很多公司机构使用，用于实验或者生产部署环境。本书将使用 TensorFlow 介绍强化学习的算法。为了帮助读者更好、更快地了解 TensorFlow 及其相关内容，本章介绍 TensorFlow 的基本操作及其背后涉及的部分原理知识，其中包括 TensorFlow 的基本使用方法、TensorFlow 框架中的一些实现细节，以及分布式的训练框架。

4.1 TensorFlow 的基本使用方法

虽然 TensorFlow 是一款流传度很广的框架，但笔者还是要用一定篇幅介绍它的基本使用方法。本章的示例代码使用 TensorFlow 1.3 及以上版本，TensorFlow 的安装比较简单，这里不再赘述，我们直接进入功能操作的介绍。以一个两层的神经网络模拟一个异或运算为例，介绍使用 TensorFlow 进行机器学习的基本步骤，其中涉及计算图准备、计算安排和执行计算三部分。

首先介绍的是问题背景。对异或运算来说，输入是两个逻辑变量，输出是一个逻辑变量。当两个输入都是 0 时，输出为 1；其他情况下均输出 0。假设我们并不清楚它的

仅供非商业用途或交流学习使用

逻辑运算方法，同时假设变量是一个浮点数，这样我们的问题就有 4 条训练数据，它的输入是：

```
data_x = np.array([[0,0],[0,1],[1,0],[1,1]], dtype=np.float32)
```

输出为：

```
data_y = np.array([[0],[1],[1],[0]], dtype=np.float32)
```

下面我们开始计算图准备的部分。首先定义模型的输入，这里将用到 **placeholder** 这个结构。它会定义一些占位变量，这些变量在定义计算图时并不会被明确指定，它的数值将在执行计算图时才被赋值，所以这个结构一般被用于传入网络的数据。我们可以从上面的输入数据中看到数据的维度为 2，所以可以定义输入变量为：

```
import tensorflow as tf
x = tf.placeholder(tf.float32, shape=[None, 2], name='input')
```

从代码中可以很明显地看出每个参数的含义，其中 shape = [None, 2] 表示在定义占位符时只知道输入的第二维是 2，第一维的大小并不确定，我们可以根据实际需要灵活地传入不同大小的数据。

接下来我们定义网络结构。我们要定义一个两层的神经网络，它的结构如图 4-1 所示。

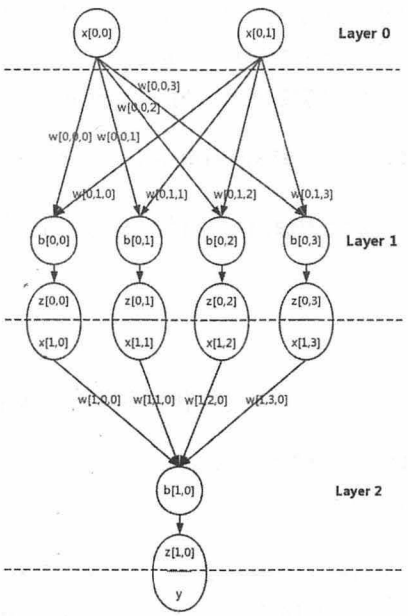


图 4-1 两层的神经网络结构

仅供非商业用途或交流学习使用

从图 4-1 中可以看出网络参数和运算细节。网络第一层的输出维度为 4，第二层的输出维度为 1。在定义神经网络的运算前，我们要先定义网络的参数变量。参数可以用 **Variable** 这个变量表示，它和前面提到的 placeholder 不同，在计算图执行前，这些变量需要被事先赋值，同时这个变量可以在计算过程中被更新。一般来说，我们有两种创建变量的方法，我们用第一种方法定义第一层的变量 w1 和 b1：

```
w1 = tf.Variable(tf.random_uniform([2,4], -1, 1), name='w1')
b1 = tf.Variable(tf.zeros(4, dtype=np.float32), name='b1')
```

可以看出这种方法直接使用了 Variable 变量进行构造。我们再用另一种方法定义第二层的变量 w2 和 b2：

```
w2 = tf.get_variable('w2', shape=[4,1], initializer=tf.contrib.layers.
xavier_initializer())
b2 = tf.get_variable('b2', shape=[1], initializer=tf.constant(0))
```

两种定义变量方法的区别将在 4.2 节介绍。定义好变量后，再来定义这些变量的运算。这里将用到一些定义好的运算函数，具体过程如下：

```
z1 = tf.sigmoid(tf.matmul(x, w1) + b1)
z2 = tf.sigmoid(tf.matmul(z1, w2) + b2)
```

从名字可以看出 matmul 和 sigmoid 方法分别代表了矩阵乘法和非线性计算两个操作，实际上每一行代码实现了一层网络的计算过程。至此，我们完成了模型部分的定义，接下来定义目标函数，我们需要再定义另一个 placeholder，用于传入目标值：

```
y = tf.placeholder(tf.float32, shape=[None, 1], name='output')
```

对于这个问题，我们直接比较输出值和目标值的差距，这个差距可以用 l2_loss 函数表示：

```
loss = tf.nn.l2_loss(z2 - y)
```

完成目标函数的设定后，接下来定义优化器和优化策略：

```
opt = tf.train.GradientDescentOptimizer(0.05)
train_op = opt.minimizer(loss)
```

我们定义了一个最基本的梯度下降法，它的学习率是 0.01。调用它的 minimize 方法就可以完成运算的定义，它的返回值 train_op 就是开启优化流程的钥匙，我们只要给两个 placeholder 赋值，并请求计算 train_op，就可以完成一轮优化。

仅供非商业用途或交流学习使用

到这里我们就完成了计算图的定义，回顾刚才的过程可以看出，代码的全过程完成了机器学习的三个核心部分的定义：定义模型、定义目标函数和定义优化方法。下面就要对计算过程进行安排。每一个计算过程都要放置在特定的计算资源上，具体来说就是 CPU 和 GPU 上，现在的计算机大多是多 CPU 核和多 GPU 的，有些情况下需要把计算安置在特定的计算资源上，这就需要我们对计算做指定。当然，即使我们不对计算过程做明确的指定，TensorFlow 也会对每一个计算安排一个计算资源。安排资源的方法为：

```
with tf.device('')
```

我们在引号内填入具体的设备编号，同时将前面提到的定义放置在这句代码之内，就完成了计算的安排。一般来说，TensorFlow 拥有一套策略用于判断怎样的安排方式可以尽可能地保证运算速度，所以我们并不需要对运算做太多的安排，这部分可操作的内容也相对较少。

最后一步就是完成真正的计算。计算前首先要创建一个变量 **Session**，它一般被翻译为会话，这里也可以认为是开启一次与计算图交互的过程，这个 Session 中会保存计算图中的变量信息，用以完成真正的计算。我们可以把前面的工作想象成“写代码”，而开启一个 Session 就是执行刚才写好的代码。开启 Session 的方法为：

```
with tf.Session() as sess:
```

后面的代码将在这句代码之内实现。下面我们要做的第一步是完成对所有变量的初始化。有的读者会问，在构建计算图时，我们已经对变量的初始值做了定义，为什么还要进行一次初始化呢？实际上，前面的初始化只是声明了变量对应的初始值，并没有真正执行赋值。这里我们要做的就是执行赋值：

```
sess.run(tf.global_variables_initializer())
```

完成了这一步，我们就可以进行真正的计算，也就是训练模型，进行多轮的梯度下降法迭代。每一轮迭代时，除了执行训练运算，还要计算并输出 loss 值，这样可以更好地监控模型的训练进展情况：

```
for iter in range(100000):
    loss_val, _=sess.run([loss, train_op], feed_dict={x:data_x, y:data_y})
    if iter % 10000 == 0:
        print('{}:loss={}'.format(iter, loss_val))
```

虽然代码中只提到了执行 train_op 这个操作，但实际上 TensorFlow 会执行所有与它有关的操作，包括神经网络的前后向计算和参数更新，所以在完成计算后，我们就完

仅供非商业用途或交流学习使用

成了模型参数的训练。如果读者对被执行的操作感兴趣，可以通过访问计算图寻找这些操作。虽然我们没有具体声明某个计算图，但这个计算图还是在我们定义过程中悄悄构建好了，只要访问 TensorFlow 为我们保存的默认计算图：

```
graph = tf.get_default_graph()
print graph.get_operations()
```

就可以看到 TensorFlow 为我们构建的一张计算图。除此之外，我们还可以使用 TensorFlow 生态中的 TensorBoard 将这个计算图以图画的形式显示出来，这部分内容将在后面章节中介绍。完成了模型的训练，将模型的参数保存下来，就完成了这个训练的过程。

接下来我们将介绍更多关于 TensorFlow 的复杂应用，想要对 TensorFlow 有更深刻的了解，还需读者不断地练习和体会。

4.2 TensorFlow 原理介绍

4.1 节介绍了 TensorFlow 的基本使用方法，本节将介绍 TensorFlow 的内部原理，同时对编程过程提出一些建议。不同于其他书籍简单地介绍每一个函数的功能和直接使用方式，笔者将从代码片段入手，通过分析代码的原理理解 TensorFlow 中的一些特性。

读者也许会问，为什么要把精力放在研究这种鸡毛碎皮的小事上，而不是直接介绍使用 TensorFlow 解决问题的案例？笔者认为，这种深入代码内部的工作确实不是必需的，但它却是对框架深入理解的基础。当我们想打开 TensorBoard 看 Graph 可视化图时，当程序抛出某个变量有问题（维度不对、未参与梯度计算等）时，理解命名规则等细节能帮你快速定位错误，修炼好内功可以让你走得更远！

前面我们已经介绍过，TensorFlow 是一个基于计算图的运算框架，它的核心操作分为三步。

（1）构建计算图。

（2）分发计算任务。

（3）执行计算任务。

对大多数的任务来说，第 2 步和第 3 步操作对于用户来说相对容易，于是我们的主要精力都花在第 1 步——也就是构建计算图上。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

4.2.1 创建变量的 scope

4.1 节介绍了创建变量的方法，这个过程中一个十分重要的环节就是为每一个变量命名。除此之外，我们还可以给每一个操作创建名字。为了更好地组织变量和操作的名字，需要使用层次结构定义名字，此时就需要 scope 这个概念。如果一个模型中包含多个全连接层和卷积层的计算，我们不能把它们的名字都定义为“fc”或者“conv”，也不应该将所有的操作按出现顺序依次起名，如将第 18 个出现的卷积操作称为“conv18”，这样做不便于定位对应的操作。就像我们在使用计算机时会用文件夹管理存储的文件一样，模型有时也需要类似的组织形式，比如名字为“vf/block2/conv2”可以让我们大概猜测出这个操作来自于值函数模型（value function model）中第 2 个运算组中的第 2 个卷积操作。这样当这个操作出现问题时，我们可以快速完成定位。定义上面提到的层次型名字就需要 scope 操作。

scope 对分离变量的命名空间有十分重要的作用，为了更好地理解 scope 的用法，我们将通过几段代码展示它的用法。首先是第一段代码：

```
import tensorflow as tf

with tf.name_scope('123'):
    with tf.name_scope('456'):
        with tf.variable_scope('789'):
            a = tf.Variable(1,name='a')
            print a.name
            b = tf.get_variable('b',1)
            print b.name
```

请读者思考，代码中的两句输出分别是什么？如果一个人号称精通 TensorFlow，那么这个问题应该可以答上来。当然，像这类问题即使猜也可以猜得八九不离十。执行代码可以得到：

```
123/456/789/b:0
789/a:0
```

从结果可以看出，对直接构建的 Variable 变量来说，每一个 scope 都会对名字产生影响；而对使用函数 get_variable 创建的变量来说，并不是所有的 scope 都会被它考虑在内。只有 variable_scope 会对它的名字产生影响，而 name_scope 不会。这也可以看出两种创建变量方法对 scope 的反应。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

下面我们增加问题的难度，如下代码（后文称第二段代码）会输出什么样的结果呢？

```
import tensorflow as tf

with tf.name_scope('123'):
    with tf.name_scope('456'):
        with tf.variable_scope('789'):
            a = tf.Variable(1,name='a')
            print a.name
            b = tf.get_variable('b',1)
            print b.name
    with tf.name_scope('456'):
        with tf.variable_scope('789'):
            c = tf.Variable(1,name='c')
            print c.name
            d = tf.Variable(1,name='d')
            print d.name
```

这段代码比较复杂，前面两个输出语句与第一段代码相同，后面两句该输出什么呢？执行这段代码可以得到：

```
123/456_1/789/c:0
123/456_1/789/d:0
```

从结果可以看出，在同一个 scope 内，同样名字的 name_scope 被声明第二次时，scope 的名字并不会直接被复用出现，而是通过改名的形式创建一个全新的 scope。

看到这里读者一定会想，name_scope 重复出现时，系统会自动改名，那么 variable_scope 呢？于是我们有了第三段代码：

```
import tensorflow as tf

with tf.name_scope('123'):
    with tf.name_scope('456'):
        with tf.variable_scope('789'):
            d = tf.Variable(1,name='d')
            print d.name
        with tf.variable_scope('789'):
            e = tf.get_variable('e',1)
            print e.name
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

当同一个 scope 下出现两遍 variable_scope 时，scope 的名字会不会出现名称修改的情况呢？这个问题留给读者，请读者亲自尝试。看到这里，相信读者对这两种 scope 有了更深的理解，下面就来看看这两个 scope 创建的实现细节。

1. name_scope

为了探究原理，我们需要去源代码中寻找答案。在 TensorFlow 源代码 tensorflow/tensorflow/python/framework/ops.py 中，我们可以找到 name_scope 函数的定义，抛开一些与核心功能无关的代码，这个函数首先要找到框架中的默认计算图（Default Graph），再调用这个默认计算图的 name_scope 方法生成 scope。

```
g = _get_graph_from_inputs(values)
with g.as_default(), g.name_scope(name) as scope:
    yield scope
```

进一步跟进代码，我们会来到 Graph 这个类的 name_scope 方法中，这个方法是一个基于 contextlib 的方法。如果传入的 scope name 在结尾包含‘/’，把这个符号删除，就可以作为命名空间的名字了。如果不是这样，就要进入一个叫作 unique_name 的方法中，和自己的父级别的命名空间的名字拼起来，在名字库中比对。如果有重名，就采用“原名_%d”的方式进行修改并尝试是否重名，直到不再重名为止。在上面的示例代码中我们看到“456_1”这样的名字，就是这个原因。有关这部分的代码如下：

```
try:
    old_stack = self._name_stack
    if not name: #Both for name=None and name="" we re-set to empty scope.
        new_stack = None
    elif name and name[-1] == "/":
        new_stack = _name_from_scope_name(name)
    else:
        new_stack = self.unique_name(name)
        self._name_stack = new_stack
        yield "" if new_stack is None else new_stack + "/"
finally:
        self._name_stack = old_stack
```

unique_name 方法的核心代码如下所示：

```
if self._name_stack:
    name = self._name_stack + "/" + name
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    i = self._names_in_use.get(name, 0)
    # Increment the number for "name".
    if mark_as_used:
        self._names_in_use[name] = i + 1
    if i > 0:
        base_name = name
        # Make sure the composed name is not already used.
        while name in self._names_in_use:
            name = "%s_%d" % (base_name, i)
            i += 1
        # Mark the composed name as used in case someone wants
        # to call unique_name("name_1").
        if mark_as_used:
            self._names_in_use[name] = 1
return name
```

看完上面的源码，我们实际上还发现了一些更奇怪的逻辑，于是可以给出一段十分诡异的代码：

```
with tf.name_scope('123'):
    with tf.name_scope(None):
        c = tf.Variable(1,name='f')
        print c.name
```

这里输出的内容又是什么样的呢？读者可以根据上面的实现代码猜猜看。

2. variable_scope

实际上，variable_scope 和 name_scope 是两个有点相互独立的命名体系，name_scope 只对通过 Variable 创建的变量有效，而 variable_scope 还可以为 get_variable 这个可以复用的变量的方法服务。那么 variable_scope 是怎么做到的呢？

我们从 tensorflow/tensorflow/python/ops/variable_scope.py 文件中找到对应的方法，文件中最核心的一段代码是这样的：

```
# name_scope就是方法传入的scope名称或者scope类对象
with ops.name_scope(name_scope) as cur_name_scope:
    if isinstance(name_or_scope, six.string_types):
        old_name_scope = cur_name_scope
    else:
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
        old_name_scope = name_or_scope.original_name_scope
    with _pure_variable_scope(
        name_or_scope,
        reuse=reuse,
        initializer=initializer,
        regularizer=regularizer,
        caching_device=caching_device,
        partitioner=partitioner,
        custom_getter=custom_getter,
        old_name_scope=old_name_scope,
        dtype=dtype,
        use_resource=use_resource,
        constraint=constraint) as vs:
            yield vs
```

可以看出，它首先调用了 name_scope 方法，然后在 name_scope 的 context 中调用了另外一个生成 variable_scope 的方法。将这个方法简化，它完成的主要工作如代码和注释所示：

```
# 1. 获取default_varscope
get_variable_scope()
default_varscope = ops.get_collection_ref(_VARSCOPE_KEY)
# 2. 获取var_store
old = default_varscope[0]
var_store = _get_default_variable_store()
# 3. 组合新scope的名字
new_name = old.name + "/" + name_or_scope
try:
    # 4. 创建新的varscope并返回
    reuse = reuse or old.reuse
    default_varscope[0] = VariableScope(... # 创建调用
    yield default_varscope[0]
finally:
    # 5. 回收scope信息
    var_store.close_variable_subscopes(new_name)
    default_varscope[0] = old
```

如注释所示，我们将整体过程分成五步，第 1 步的主要方法为 get_variable_scope，它的代码如下：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
def get_variable_scope():
    scope = ops.get_collection(_VARSCOPE_KEY)
    if scope:
        return scope[0]
    scope = VariableScope(False)
    ops.add_to_collection(_VARSCOPE_KEY, scope)
    return scope
```

它的功能其实就是创建一个全局的 var_scope，并确保 collection 中有内容。实际上，这里只有一个元素，因为 collection 设计成存储列表的形式，所以访问元素需要访问 0 号下标。下一句代码中，这个确定创建的 default_varscope 就被取出来了。

确保了 var_scope 的创建，第 2 步就是访问_get_default_variable_store 方法，它的代码如下：

```
def _get_default_variable_store():
    store = ops.get_collection(_VARSTORE_KEY)
    if store:
        return store[0]
    store = _VariableStore()
    ops.add_to_collection(_VARSTORE_KEY, store)
    return store
```

可以看到，这个函数创建了一个 var_store 对象。它将完成名字回收的工作。

在第 3 步中，代码要设定新 varscope 的名字，new_name 和当前 var_scope 的名字有关，但和我们前面看到的 name_scope 并无关系。第 4 步，代码将按照这个新名字定义 scope 的名字，并创建一个 VariableScope 对象。第 5 步则是在 scope 使用完成后回收 scope 信息。

当完成了 context 内的操作并退出 varscope 时，这种栈结构的命名空间模式会将默认的 varscope 替换为进入方法时原本的 varscope。这样，基本的代码过程就结束了。

总体来看，两种 scope 存在一些相同和不同，让我们来总结一下。

（1）两种 scope 形成的命名空间将在 Variable 创建的变量上产生影响。

（2）variable_scope 创建的命名空间将对 get_variable 的变量名产生影响。

（3）创建 named_scope 时，遇到同名的 scope，系统会自动改名创建一个新的 scope。

（4）创建 named_scope 时，可以通过将名字设置为 None 抹掉前面设定的所有命名空间的名字。

希望读者能够了解 scope 为命名带来的便利和复杂。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

4.2.2 创建一个 Variable 背后的故事

4.2.1 节介绍了 scope 的细节，本节介绍创建变量需要关注的内容。我们已经知道了两种创建变量的方法，它们究竟创造了些什么呢？我们同样从几段代码中寻找线索。

1. tf.Variable()

先来看第一段代码：

```
import tensorflow as tf
a = tf.Variable(1, name='a')
g = tf.get_default_graph()
print g.get_operations()
```

在这段代码中，我们用 Variable 方法创建了一个变量，并获取了系统的默认计算图，将计算图中的操作输出，它会输出什么呢？看上去我们只创建了一个变量，在默认的计算图上应该只有一个变量的相关操作吧？实际上并不是这样，输出结果如下所示：

```
[<tf.Operation 'a/initial_value' type=Const>,
<tf.Operation 'a' type=VariableV2>,
<tf.Operation 'a/Assign' type=Assign>,
<tf.Operation 'a/read' type=Identity>]
```

我们发现系统创建了四个操作——除了一个和变量名相同但类型为“VariableV2”的操作外，还有三个操作：Const、Assign 和 Indentiy。这些操作的输入和输出又是什么呢？

我们将四个操作（Operation，以下简称 Op）的定义输出，可以看到更多细节：

```
# op[0]
name: "a/initial_value"
op: "Const"
attr {
  key: "dtype"
  value {
    type: DT_INT32
  }
}
attr {
  key: "value"
  value {
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    tensor {
      dtype: DT_INT32
      tensor_shape {
      }
      int_val: 1
    }
  }
}
# op[1]
name: "a"
op: "VariableV2"
attr {
  key: "container"
  value {
    s: ""
  }
}
attr {
  key: "dtype"
  value {
    type: DT_INT32
  }
}
attr {
  key: "shape"
  value {
    shape {
    }
  }
}
attr {
  key: "shared_name"
  value {
    s: ""
  }
}
# op[2]
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
name: "a/Assign"
op: "Assign"
input: "a"
input: "a/initial_value"
attr {
  key: "T"
  value {
    type: DT_INT32
  }
}
attr {
  key: "_class"
  value {
    list {
      s: "loc:@a"
    }
  }
}
attr {
  key: "use_locking"
  value {
    b: true
  }
}
attr {
  key: "validate_shape"
  value {
    b: true
  }
}
# op[3]
name: "a/read"
op: "Identity"
input: "a"
attr {
  key: "T"
  value {
```

仅供非商业用途或交流学习使用


```
    type: DT_INT32
  }
}
attr {
  key: "_class"
  value {
    list {
      s: "loc:@a"
    }
  }
}
```

对照着输出的结果，我们再来看 tensorflow/python/ops/variables.py 中创建变量的缩减版代码和注释就很清楚了：

```
def _init_from_args(self, ...
    with ops.name_scope(name) as name:
        # 1. 将初始值转成Tensor
        self._initial_value = ops.convert_to_tensor(
              initial_value, name="initial_value", dtype=dtype)
        shape = (self._initial_value.get_shape()
                 if validate_shape else tensor_shape.unknown_shape())
        # 2. 创建Variable
        self._variable = state_ops.variable_op_v2(
              shape, self._initial_value.dtype.base_dtype,
              name=name)
        # 3. 将初始值赋值给Variable
        self._initializer_op = state_ops.assign(
            self._variable,
            self._build_initializer_expr(self._initial_value),
            validate_shape=validate_shape).op
        # 4. 将变量的值读取出来，变成可操作的对象
        self._snapshot = array_ops.identity(self._variable, name="read")
```

如代码中的注释所示，创建操作的工作就是在这里完成的，整个过程分为四步。

（1）将初始值变为一个 Tensor。

（2）创建一个变量。

（3）将初始值赋值给变量。

仅供非商业用途或交流学习使用

（4）将变量的值读取出来，变成可以被后续操作使用的对象。

当我们想进一步追踪这些方法背后的调用时，会发现这些方法基本被指向了一些C++的代码，由于C++里面的内容会更复杂，这里我们暂时不去触碰这些代码。通过这段代码分析我们发现，创建一个变量的工作是由好几个操作完成的，这能够帮助我们更好地理解Python代码和真实计算图之间的关系。

2. tf.get_variable()

看完了第一种创建变量的方法，我们再来看另一种创建变量的方法，对应的代码为：

```
import tensorflow as tf
a = tf.get_variable('a',1)
g = tf.get_default_graph()
print g.get_operations()
```

它会不会和前面一样产生四个Op呢？经过执行发现，实际上它得到了更多的Op：

```
[<tf.Operation 'a/Initializer/random_uniform/shape' type=Const>,
 <tf.Operation 'a/Initializer/random_uniform/min' type=Const>,
 <tf.Operation 'a/Initializer/random_uniform/max' type=Const>,
 <tf.Operation 'a/Initializer/random_uniform/RandomUniform' type=
 RandomUniform>,
 <tf.Operation 'a/Initializer/random_uniform/sub' type=Sub>,
 <tf.Operation 'a/Initializer/random_uniform/mul' type=Mul>,
 <tf.Operation 'a/Initializer/random_uniform' type=Add>,
 <tf.Operation 'a' type=VariableV2>,
 <tf.Operation 'a/Assign' type=Assign>,
 <tf.Operation 'a/read' type=Identity>]
```

从名字上看，方法生成的Op总体上和上一个方法类似，只不过它不是直接用数值初始化，而是使用一个初始化的方法，因此我们需要设定这个初始化方法的参数。下面需要关注的是变量的正则项：

```
if regularizer:
    with ops.colocate_with(v):
        with ops.name_scope(name + "/Regularizer/"):
            loss = regularizer(v)
            if loss is not None:
                if context.in_graph_mode():
                    v_name = v.name
```

仅供非商业用途或交流学习使用


```
            loss_name = loss.name
        else:
            v_name = "v_%s" % type(v)
            loss_name = "loss_%s" % type(loss)
        ops.add_to_collection(ops.GraphKeys.REGULARIZATION_LOSSES,
            loss)
```

从这部分内容可以看出，如果在创建变量时设置了正则项，那么正则项将被加入 collection 中，我们可以根据需要将其取出并加入 loss 中。

经过分析我们发现，虽然我们在创建变量时只写了一句代码，背后却产生了很多更细粒度的 Op，所以在计算图执行时，我们要明白这些 Op 的来源。

4.2.3 运算操作

从上面的代码中我们已经看出，在创建变量时，我们的默认 Graph 实际上创建了许多 Op，这些 Op 组成了整个计算图。执行计算图就是执行其中的 Op。那么每一个 Op 又包含什么属性呢？在探索更复杂的内容之前，我们先来看看组成 Graph 的基石——Op 更多的细节。

我们来看一段简单的代码：

```
import tensorflow as tf
a = tf.get_variable('a',1)
b = tf.get_variable('b',2)
c = a + b
g = tf.get_default_graph()
print g.get_operations()[-1]
```

代码对应的输出为：

```
<tf.Operation 'add' type=Add>
```

当我们完成了一个加法运算的声明，图中就多出了这个“add”Op。这个 Op 拥有自己的 op_def 方法，显示出来是这样的：

```
name: "Add"
input_arg {
  name: "x"
  type_attr: "T"
}
```

仅供非商业用途或交流学习使用


```
input_arg {
  name: "y"
  type_attr: "T"
}
output_arg {
  name: "z"
  type_attr: "T"
}
attr {
  name: "T"
  type: "type"
  allowed_values {
    list {
      type: DT_HALF
      type: DT_FLOAT
      type: DT_DOUBLE
      type: DT_UINT8
      type: DT_INT8
      type: DT_INT16
      type: DT_INT32
      type: DT_INT64
      type: DT_COMPLEX64
      type: DT_COMPLEX128
      type: DT_STRING
    }
  }
}
```

从这里可以看到有关 Op 的所有信息。我们还可以获得有关它的输入和输出的定义：

```
In:  add.inputs[:]
Out: [<tf.Tensor 'a/read:0' shape=(1,) dtype=float32>,
      <tf.Tensor 'b/read:0' shape=(2,) dtype=float32>]
In:  add.outputs
Out: [<tf.Tensor 'add:0' shape=(2,) dtype=float32>]
```

可以看出，虽然在代码中相加之后赋值的变量为 c，但是在计算图中，它把相加输出的值叫作“add:0”。如果读者不了解这个原理，那么看到这个名字也许会发懵。这也说明在平时编程时，为每一个 Op 起名的重要性。如果将中间的代码写成：

仅供非商业用途或交流学习使用


```
c = tf.add(a, b,'c')
```

那么它的输出就会变成：

```
In:  add.outputs
Out: [<tf.Tensor 'c:0' shape=(2,) dtype=float32>]
```

虽然它不叫作 c，但至少与 c 有关，我们可以比较方便地查找定位它。如果我们想从计算图中找到这个代表输出的 Tensor，可以采用下面的方法：

```
In:  g.get_tensor_by_name('c:0') # 利用Graph的方法进行查找
Out: <tf.Tensor 'c:0' shape=(2,) dtype=float32>
```

通过上面的分析，我们发现代码和计算图存在一定的不同，为了编程的方便，一些繁杂的设定被系统以默认值替代，但有时会为我们带来麻烦，如果前面的加法出现错误（例如参数运算的 Tensor 维度不一致），那么'add:0' 一定不如'c:0' 容易定位，所以要尽可能地为每一个 Op 起好名字。

4.2.4 tf.gradients

前面我们介绍了与变量、操作相关的一些知识，下面我们介绍关于反向计算的一些细节。在介绍神经网络时，我们一直提到“前向计算”和“反向计算”这两个词，这容易给读者留下一个印象：前向计算和后向计算的计算是在同一批计算节点上完成的，只不过计算的方向不同。但实际上真的如此吗？如果不是这样，那么反向计算的结构是什么样的呢？我们来看看下面的代码：

```
import tensorflow as tf
a = tf.constant(1.0)
b = tf.constant(1.0)
c = a + b
g = tf.get_default_graph()
print g.get_operations()
grad = tf.gradients(c, [a,b])
print g.get_operations()
```

这段代码的输出结果为：

```
[<tf.Operation 'Const' type=Const>,
<tf.Operation 'Const_1' type=Const>,
<tf.Operation 'add' type=Add>]
```

仅供非商业用途或交流学习使用


```
[<tf.Operation 'Const' type=Const>,
<tf.Operation 'Const_1' type=Const>,
<tf.Operation 'add' type=Add>,
<tf.Operation 'gradients/Shape' type=Const>,
<tf.Operation 'gradients/Const' type=Const>,
<tf.Operation 'gradients/Fill' type=Fill>,
<tf.Operation 'gradients/add_grad/Shape' type=Const>,
<tf.Operation 'gradients/add_grad/Shape_1' type=Const>,
<tf.Operation 'gradients/add_grad/BroadcastGradientArgs' type=
BroadcastGradientArgs>,
<tf.Operation 'gradients/add_grad/Sum' type=Sum>,
<tf.Operation 'gradients/add_grad/Reshape' type=Reshape>,
<tf.Operation 'gradients/add_grad/Sum_1' type=Sum>,
<tf.Operation 'gradients/add_grad/Reshape_1' type=Reshape>]
```

经过梯度计算，我们增加了许多新的 Op。这些 Op 是用来做什么的？它们之间是什么关系？因为涉及的节点比较少，所以可以用 TensorBoard 将计算图画出来。执行下面的代码：

```
tf.summary.FileWriter("logs", g).close()
```

这样计算图就会绘制在我们指定的目录 logs 中。再利用 TensorBoard 显示绘制的信息：

```
tensorboard --logdir=logs
```

就可以从显示的页面中看到如图 4-2 所示的图像。

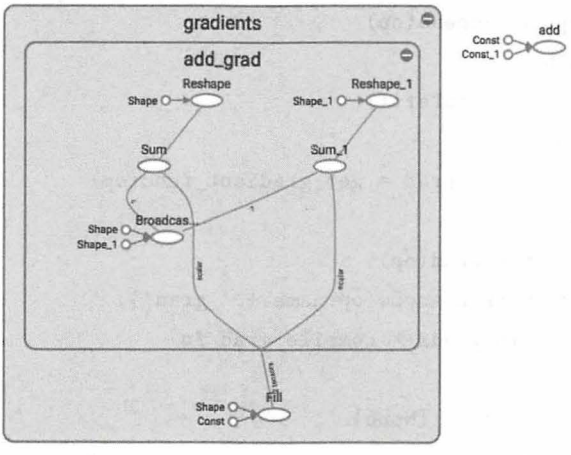


图 4-2 代码对应的计算图

仅供非商业用途或交流学习使用

我们可以看到图 4-2 由两部分组成，右边是一个很小的图，这部分是前向计算的图，左边是一个很大的图，外层的名字为 gradients，内层的名字为 add_grad，add_grad 中的内容就是右边 add 操作的反向计算部分。那么，这么庞大的内容是如何生成的呢？与这部分内容有关的代码在/tensorflow/tensorflow/python/ops/gradients_impl.py 中，下面就来看看它的基本流程，代码和注释如下所示：

```
def gradients(ys, xs):
    with name_scope('gradients'):
        # step 1
        grad_ys = _DefaultGradYs(grad_ys, ys)
        to_ops = [t.op for t in ys]
        from_ops = [t.op for t in xs]
        # step 2
        pending_count = _PendingCount(to_ops, from_ops)
        # step 3
        grads = {}
        for y, grad_y in zip(ys, grad_ys):
            _SetGrad(grads, y, grad_y)
        # step 4
        queue = collections.deque()
        to_ops_set = set()
        for op in to_ops:
            ready = (pending_count[op._id] == 0)
            if ready and op._id not in to_ops_set:
                to_ops_set.add(op._id)
                queue.append(op)
        while queue:
            op = queue.popleft()
            # step 5
            grad_fn, out_grad = get_gradient_func(op)
            # step 6
            if has_out_grad(op):
                with name_scope(op.name + '_grad'):
                    in_grads = compile grad_fn
            else:
                in_grads = [None]
            # step 7
            for t_in, in_grad in zip(op.inputs, in_grads):
```

仅供非商业用途或交流学习使用


```
                _SetGrad(grads, t_in, in_grad)
            # step 8
            _UpdatePendingAndEnqueueReady(grads, op, queue, pending_count)
    # step 9
    return [_GetGrad(grads, x) for x in xs]
```

函数的输入分别为代求导的输出值 ys 和输入值 xs，对于上面的这段代码，我们可以将整个构建梯度计算图的过程分解成下面 9 个步骤。

第 1 步（Step 1）：为每个求导的 y 赋值一个默认导数，这段代码将在 Graph 中加入'gradients/Fill' 这个 Op，也就是默认的梯度值 1；同时，将反向计算求导的起点和终点的 Op 保存下来，其中 to_ops 保存了起点，也就是 ys 中的 Op，from_ops 保存了终点，也就是 xs。反向计算就是从 to_ops 出发计算梯度，直到我们将所有 from_ops 的梯度计算完成。

第 2 步（Step 2）：根据计算图的结构，从 to_ops 方向向 from_ops 计算 pending_count 数组。这个数组记录了 Graph 中每一个节点在计算梯度时依赖的节点数目，一个 Op 有多少个输出的 Op，就有多少个依赖。当 pending_count 中节点对应的值为 0 时，说明它依赖的前置梯度已经计算完成，可以计算它的梯度了。

第 3 步（Step 3）：将现在的梯度保存在一个用于存储 Op 与对应梯度 Op 关系的 map 中，这个 map 被称为 grads。

第 4 步（Step 4）：遍历 to_ops，将可以进行梯度计算的 Op（pending_count=0）放入队列（queue）中。

第 5 步（Step 5）：从 queue 中取出 Op，从 TensorFlow 的系统中找到对应的反向 Op。TensorFlow 会事先记录好一个 Op 和对应的反向 Op 的关系，使用时只需要按名字查找即可。

第 6 步（Step 6）：如果找到了反向 Op，那么执行编译工作将生成反向 Op；如果没有，则返回 None。

第 7 步（Step 7）：将第 6 步新生成的 Op 保存到 grads 中。

第 8 步（Step 8）：更新 pending_count，将更多满足求导条件的 Op 放入 queue 中。

第 9 步（Step 9）：将待求的 Tensor 的梯度 Op 从 grads 中查找出来，作为返回。

在这些步骤中，第 2 步的过程相对复杂，也比较重要，它基于计算图构建了后面几步工作的基础：每一个 Op 计算梯度时的依赖数——pending_count。这一步需要对计算图进行多次遍历，对应的简化版代码如下所示：

```
def _PendingCount(to_ops, from_ops):
```

仅供非商业用途或交流学习使用


```
    # step 2.1: 过滤to_ops和from_ops中间涉及梯度计算的所有Op
    reached_ops = [False] * (graph_node_size)
    for op in to_ops:
        reached_ops[op._id] = True
    # step 2.2: 从from_ops出发，访问每一个Op的输出，并将reached_ops中的输
    出节点标记为True，依此遍历所有经过的节点
    _MarkReachedOps(from_ops, reached_ops)
    between_ops = [False] * (graph._last_id + 1)
    between_op_list = []
    queue = collections.deque()
    queue.extend(to_ops)
    # step 2.3 反向遍历节点并将to_ops和from_ops中间的节点记录在between_ops
    中
    while queue:
        op = queue.popleft()
        if reached_ops[op._id]:
            between_ops[op._id] = True
            between_op_list.append(op)
            reached_ops[op._id] = False
            for inp in op.inputs:
                queue.append(inp.op)
    pending_count = [0] * (graph._last_id + 1)
    # step 2.4 将between_ops中的ops的pending_count标记出来
    for op in between_op_list:
        for x in op.inputs:
            if between_ops[x.op._id]:
                pending_count[x.op._id] += 1
    return pending_count
```

上面的代码看得有些复杂，我们来举一个例子看看它的计算方法。假设有图 4-3 所示的计算图，其中包含每个节点被访问的情况，待求的是 to_ops 和 from_ops。

第 2.1 步，我们将 to_ops 中的节点标记为 reached，也就是说它已经被访问了，如图 4-4 所示。

第 2.2 步，从 from_op 出发，将正向的节点标记为 reached，如图 4-5 所示。

第 2.3 步，从 to_ops 出发，反向遍历，将节点由 reached 标记改为 between 标记，如图 4-6 所示。

仅供非商业用途或交流学习使用

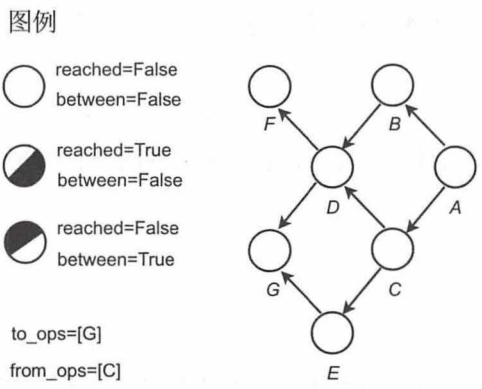


图 4-3 计算图的基本结构

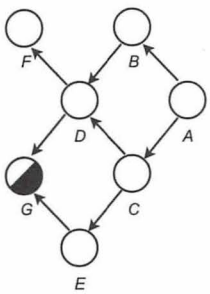


图 4-4 第 2.1 步：计算后的结果

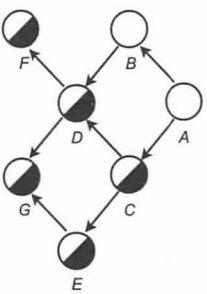


图 4-5 第 2.2 步：计算后的结果

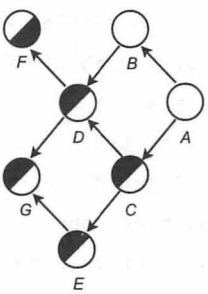


图 4-6 第 2.3 步：计算后的结果

第 2.4 步，遍历由 between 标记的节点，记录 pending_count，对于前面所举的例子来说，它的数值为：

```
pending_count[G]=0
pending_count[D]=1
pending_count[E]=1
pending_count[C]=2
```

了解了这部分的功能，我们来看最终得到的 grads 变量的内容为：

```
{<tf.Operation 'add' type=Add>: [<tf.Tensor 'gradients/Fill:0' shape=()
dtype=float32>],
 <tf.Operation 'Const' type=Const>: [[<tf.Tensor 'gradients/add_grad/
 Reshape:0' shape=() dtype=float32>]],
 <tf.Operation 'Const_1' type=Const>: [[<tf.Tensor 'gradients/add_grad/
 Reshape_1:0' shape=() dtype=float32>]]}
```

仅供非商业用途或交流学习使用

我们将这里的 Op 和图 4-2 中的名字进行对应，就可以更深入地了解反向计算图的构造。

通过对这部分源码的分析，相信大家对 TensorFlow 构建反向计算有了一定的了解，可以看出反向计算比前向计算更复杂，也容易出现问题，因此了解一点细节对读者大有裨益。

4.2.5 Optimizer

4.2.4 节我们了解了反向计算的细节，本节介绍优化部分计算图的构建过程。我们来看一个简单的例子：

```
import tensorflow as tf
a = tf.Variable(1.0 ,name='a')
loss = a - 1
opt = tf.train.GradientDescentOptimizer(0.1)
g = tf.get_default_graph()
print g.get_operations()
grad = opt.compute_gradients(loss)
print g.get_operations()
train_op = opt.apply_gradients(grad)
print g.get_operations()
tf.summary.FileWriter("logs", g).close()
```

上面代码的三句输出分别输出了下面的内容：

```
[<tf.Operation 'a/initial_value' type=Const>,
<tf.Operation 'a' type=VariableV2>,
<tf.Operation 'a/Assign' type=Assign>,
<tf.Operation 'a/read' type=Identity>,
<tf.Operation 'sub/y' type=Const>,
<tf.Operation 'sub' type=Sub>]

(这里省略相同的内容……)
<tf.Operation 'gradients/Shape' type=Const>,
<tf.Operation 'gradients/Const' type=Const>,
<tf.Operation 'gradients/Fill' type=Fill>,
<tf.Operation 'gradients/sub_grad/Shape' type=Const>,
<tf.Operation 'gradients/sub_grad/Shape_1' type=Const>,
```

仅供非商业用途或交流学习使用


```
<tf.Operation 'gradients/sub_grad/BroadcastGradientArgs'
type=BroadcastGradientArgs>,
<tf.Operation 'gradients/sub_grad/Sum' type=Sum>,
<tf.Operation 'gradients/sub_grad/Reshape' type=Reshape>,
<tf.Operation 'gradients/sub_grad/Sum_1' type=Sum>,
<tf.Operation 'gradients/sub_grad/Neg' type=Neg>,
<tf.Operation 'gradients/sub_grad/Reshape_1' type=Reshape>,
<tf.Operation 'gradients/sub_grad/tuple/group_deps' type=NoOp>,
<tf.Operation 'gradients/sub_grad/tuple/control_dependency' type=
Identity>,
<tf.Operation 'gradients/sub_grad/tuple/control_dependency_1' type=
Identity>]

(这里省略相同的内容……)
<tf.Operation 'GradientDescent/learning_rate' type=Const>,
<tf.Operation 'GradientDescent/update_a/ApplyGradientDescent' type=
ApplyGradientDescent>,
<tf.Operation 'GradientDescent' type=NoOp>]
```

最终生成的计算图如图 4-7 所示。

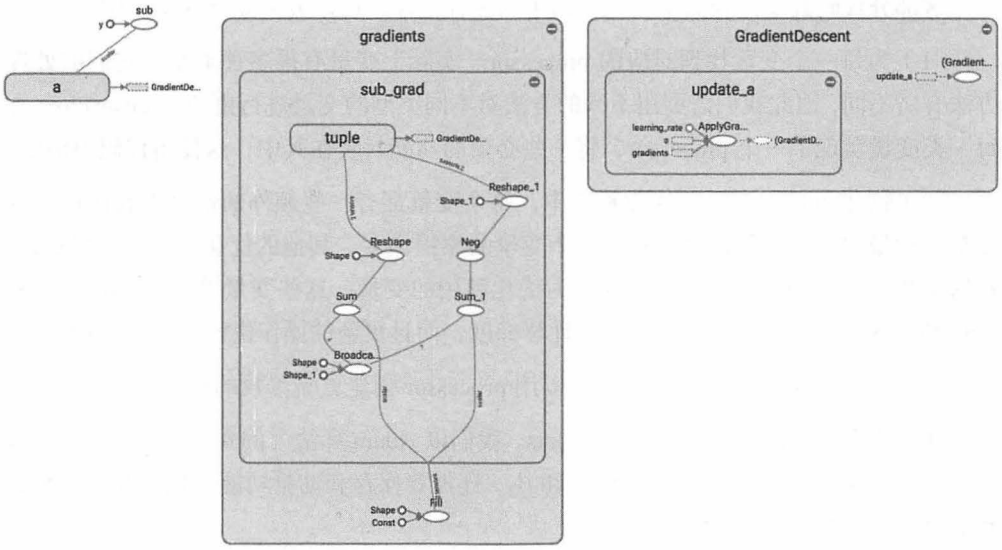


图 4-7 代码对应的计算图

从图 4-7 中可以看出，除了前面已经知道的用于反向计算的部分外，Optimizer 还

仅供非商业用途或交流学习使用

为计算图增加了一部分。这部分增加的内容由 apply_gradients 函数生成，它的主要结构如下：

```
def apply_gradients(grads_and_vars):
    # step1: 为每一个变量找到对应的processor
    converted_grads_and_vars = []
    for g,v in grads_and_vars:
        p = _get_processor(v)
        converted_grads_and_vars.append((g, v, p))
    # step2: 创建优化器必需的slot变量
    _create_slots(var_list)
    # step3: 用processor创建更新参数的计算
    with name_scope(opt_name):
        update_ops = []
        for grad, var, processor in converted_grads_and_vars:
            with name_scope("update_"+ var_name):
                update_op = processor.update_op(grad)
                update_ops.append(update_op)
        apply_updates = _finish(update_ops)
    return apply_updates
```

这部分代码其实比较简单，如代码中的注释一样，可以分为如下 3 个步骤。

（1）为每一个变量找到对应的 processor。实际上变量有很多种类型，它们的处理方法有所不同，因此我们需要用不同的方法对不同类型的变量进行更新。TensorFlow 为每一类变量实现了一个 processor，将一些必要的功能封装在其中，这使得代码更清晰。

（2）创建 slots。在一些优化算法中，每个变量都有一些额外的信息需要保存，例如基于动量的方法，我们要记录每一个变量的累积梯度。其他的优化算法也同样如此，在这里我们可以为它们创建了一些专为优化私用的变量，这些变量被称为 slots。这些 slots 和普通的变量不同，它们不需要计算梯度，而且更新的操作由优化方法自行定义。

（3）获得 processor 后，我们就可以用 processor 创建更新参数的计算 Op。

这里值得一提的是第 2 步：创建 slots。我们以 Adam 算法为例介绍如何实现它。这个算法需要保存一些参数变量，如 β_1 和 β_2，还需要保存和动量功能类似的变量，如 m 和 t。它的主要优化流程为

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2) g_t^2$$

仅供非商业用途或交流学习使用

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$x_{t+1} = x_t - \mathrm{lr}\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}$$

可以看出，为了算法可以执行，优化器中需要维护这些积累变量，而它们就要保存在 slots 中。Adam 的代码在 tensorflow/python/training/adam.py 中，其中就包含了创建 slots 的代码。

```
def _create_slots(self, var_list):
    self._create_non_slot_variable(initial_value=self._beta1,
                                   name="beta1_power")
    self._create_non_slot_variable(initial_value=self._beta2,
                                   name="beta2_power")
    for v in var_list:
        self._zeros_slot(v, "m", self._name)
        self._zeros_slot(v, "v", self._name)
```

可以看到，代码中一共创建了 4 个变量，其中两个是 beta1_power 和 beta2_power，它们保存了 β_1^t 和 β_2^t 这两个变量的值，另外两个是 m 和 v，它们被保存为 slots 值。

介绍完 slots 的创建过程，我们可以通过访问计算图中的变量集合找到优化器为我们添加的变量。

```
In [4]: g.get_collection('variables')
Out[4]:
[<tf.Variable 'a:0' shape=() dtype=float32_ref>,
 <tf.Variable 'beta1_power:0' shape=() dtype=float32_ref>,
 <tf.Variable 'beta2_power:0' shape=() dtype=float32_ref>,
 <tf.Variable 'a/Adam:0' shape=() dtype=float32_ref>,   # m
 <tf.Variable 'a/Adam_1:0' shape=() dtype=float32_ref>] # v

In [5]: g.get_collection('trainable_variables')
Out[5]: [<tf.Variable 'a:0' shape=() dtype=float32_ref>]
```

从结果可以看出，虽然我们只创建了一个变量，但为了优化，系统又为我们创建了 4 个变量，其中最后两个变量分别是 m 和 v，但实际上，可以训练的变量只有我们创建的 a。

仅供非商业用途或交流学习使用

4.2.6 TensorFlow 的反向传播技巧

TensorFlow 的反向传播可以根据前向运算自动计算得到，这也是 TensorFlow 的一个特点，也是很多计算图引擎的特点。但是这部分完全透明的内容并不像我们想象的那样简单，接下来笔者就介绍由论文 *Shake-Shake regularization of 3-branch residual networks* 的实现引出的一个问题：如何使用 TensorFlow 在反向传播的过程中修改某个变量的数值，从而修改最终的梯度？

我们将其转变成一个简单的问题，令函数 $f(x)=ax$，其中 a 是一个 0~1 的随机数，当 a 固定不变且 $x=1$ 时，$f(x)$ 和 $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 的值是相等的。现在我们希望计算图在分别进行前向和后向计算时，a 能够分别进行一次随机采样，得到两个不同的数，并使得 $f(x)$ 和 $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ 的最终结果不一样。也就是说，计算图执行的顺序是：

（1）随机采样得到 a。

（2）前向计算得到 $f(x)$。

（3）随机采样得到新的 a。

（4）反向计算得到 $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$。

那么这个过程该如何实现呢？我们先完成如下代码：

```
import tensorflow as tf
import numpy as np
with tf.name_scope('normal'):
    x = tf.placeholder(tf.float32, [],'x')
    a = tf.Variable(0.0, 'a')
    c = tf.random_uniform([], 0.0, 1.0)
    op = a.assign(c)
    with tf.control_dependencies([op]):
        f = a * x
        g = tf.gradients(f, x)
with tf.Session() as s:
    s.run(tf.global_variables_initializer())
    print s.run([f,g], feed_dict={x:1})
```

直接执行这段代码，我们发现 f 和 g 的输出结果是一样的，这并没有达到我们想要的效果。那么，有没有别的方法可以实现呢？我们知道 TensorFlow 中有 tf.control_dependencies 方法，我们能不能用它来控制求导的顺序，在反向计算前强制执行一次随机赋值呢？于是上面代码的核心部分变为：

仅供非商业用途或交流学习使用


```
with tf.name_scope('normal'):
    x = tf.placeholder(tf.float32, [],'a')
    a = tf.Variable(0.0, 'w')
    c = tf.random_uniform([], 0.0, 1.0)
    op = a.assign(c)
    c2 = tf.random_uniform([], 0.0, 1.0)
    op2 = a.assign(c2)
    with tf.control_dependencies([op]):
        f = a * x
        with tf.control_dependencies([op2]):
            g = tf.gradients(f, x)
```

修改后的效果如何呢？感兴趣的读者可以尝试，实际效果是：两个数字仍然相同。

这是为什么呢？实际上在 TensorFlow 求导时，除了函数输出部分的导数值可以通过传入参数修改（也就是方法中的 grad_ys 参数，默认为 1），其他部分 Tensor 和 Variable 的数值已经被确定并保存，对它们的修改不会影响最终梯度的计算，它还是会以前向计算时的数值进行计算。这使得我们无法以常规手段对变量进行修改。

对于这样的问题，我们有什么好的解决方法呢？接下来介绍两个方法。

方法 1：stop_gradient 技巧

tf.stop_gradient 方法用于告诉计算图，方法中的变量将不参与梯度计算。在反向计算时，它的梯度将直接设为 0。这个特性可以帮助我们实现上面的工作。具体的方法是：我们创建两个随机数 a_1和a_2，让前向计算时的数值等于 a_1x，让反向计算时的数值等于 a_2x，两者的差距则放入 stop_gradient 方法中。具体方法如下：

```
with tf.name_scope('normal'):
    x = tf.placeholder(tf.float32, [],'a')
    a1 = tf.random_uniform([], 0.0, 1.0)
    a2 = tf.random_uniform([], 0.0, 1.0)
    with tf.control_dependencies([op]):
        f1 = a1 * x
        f2 = a2 * x
        # 下面这句是关键
        f = f2 + tf.stop_gradient(f1 - f2)
        with tf.control_dependencies([op2]):
            g = tf.gradients(f, x)
```

仅供非商业用途或交流学习使用

我们来分析这关键的一句，在前向计算时，我们不需要考虑 tf.stop_gradient 的效果，因此前向计算的结果为 $f = f2 + f1 - f2 = f1$；而在反向计算时，tf.stop_gradient 中的内容没有梯度，所以相当于只对 $f2$ 进行求导，这样我们就实现了想要的效果，但是并没有改变前后向计算的机制。读者可以尝试一下，此时 f 和 g 输出的结果将变得不同。

方法 2：注册梯度函数

与 stop_gradient 方法不同，笔者将要介绍的方法可能显得更优雅，当然它也会带来一定的额外工作。在这个方法中，我们并不使用 TensorFlow 中的原生梯度函数，而是使用一个自己定制的梯度函数。前面例子中的运算实际上就是一个标量乘法，我们来写一个替换标量运算的梯度函数：

```
@tf.RegisterGradient("mult_grad")
def _mult_grad(op, grad):
    c2 = np.random.uniform(0.0, 1.0)
    return  op.inputs[1] * grad, c2 * grad
```

这个函数实现了乘法的梯度运算，它的输入分别为待求梯度的 Op 和 Op 输出的梯度值。对于函数 $f(x) = ax$，我们的函数应该返回两个偏导数：$\frac{\partial f(x)}{\partial a} = x$，$\frac{\partial f(x)}{\partial x} = a$，但是因为第 2 个参数需要进行一次随机，因此我们在方法中实现了这个随机。

然后就可以进行下面的运算：

```
1 g = tf.get_default_graph()
2 x = tf.placeholder(tf.float32, [])
3 a = tf.random_uniform([], 0.0, 1.0)
4 with g.gradient_override_map({"Mul":"mult_grad"}):
5     f = tf.multiply(a, x)
6     g = tf.gradients(f, x)
7 with tf.Session() as s:
8     s.run(tf.global_variables_initializer())
9     print s.run([f, g], feed_dict={x: 1})
```

在代码的第 4 行，我们要调用计算图的方法，将 Op 原本的梯度计算覆盖，换成我们想要的计算方法，这样就可以达到我们想要的效果。读者可以自行运行，查看结果是否符合预期。

通过对比可以看出，第 1 种方法实现更简单，但是逻辑上有些绕，需要读者厘清思路，同时它会增加一些额外的计算量；第 2 种方法从逻辑上更清晰，但是需要重写反向传播函数，这里存在着一定的风险，例如函数存在 Bug。当读者有这样的需求时，可以

仅供非商业用途或交流学习使用

自行选择其中任意一种方法解决这个问题。虽然 TensorFlow 留下了解决这个问题的方法，但两种方法都不够直观，这也算是 TensorFlow 设计上的一点缺陷。

4.2.7 arg_scope 的使用

TensorFlow 大而全的设计使我们能够实现更多的模型，但与此同时它也无法像一些框架那样保持小而美的优雅代码。我们一定很厌烦使用 TensorFlow 原生的卷积层方法，因为它的参数实在有些长：

```
tf.nn.conv2d(bottom, filter=[3,3,3,16], strides=[1,1,1,1], padding='SAME',
 data_format='NHWC')
```

我们更厌烦的是在一个模型里把这样的语句反复写多遍：

```
top1 = tf.nn.conv2d(bottom1, filter=[3,3,3,16], strides=[1,1,1,1], padding
='SAME', data_format='NHWC')
bottom2 = tf.nn.relu(top1)
top2 = tf.nn.conv2d(bottom2, filter=[3,3,16,32], strides=[1,1,1,1],
padding='SAME', data_format='NHWC')
bottom3 = tf.nn.relu(top2)
......
```

于是为了减少重复劳动，封装模型和默认参数就成了我们必须要做的事情。为了解决这个问题，很多优秀的第三方接口库随之诞生，arg_scope 就是其中之一，它被包含在 TensorFlow 的 contrib 文件夹中。使用这个功能，我们只需要定义一个关于参数的上下文空间，并在其中定义默认参数的赋值。对于空间内的语句，我们不需要对那些拥有相同赋值的参数再次赋值，参数空间会为我们自动完成这个工作。上面的代码用 arg_scope 的方式进行重写，就可以得到：

```
import tf.contrib.framework.arg_scope as arg_scope
with arg_scope([tf.nn.conv2d], strides=[1,1,1,1], padding='SAME',
data_format='NHWC'):
    top1 = tf.nn.conv2d(bottom1, filter=[3,3,3,16])
    bottom2 = tf.nn.relu(top1)
    top2 = tf.nn.conv2d(bottom2, filter=[3,3,16,32])
    bottom3 = tf.nn.relu(top2)
......
```

仅供非商业用途或交流学习使用

从代码中可以看出，相同赋值的参数被定义在 arg_scope 的定义中，这里所有语句的参数将被自动赋值。通过这个改变，代码变得清爽了。因为那些不变的参数只需要定义一次，所以我们只需关注那些经常变化的参数即可。

其实实现这样的功能并不复杂，这种参数共享的实现用到了 contextlib 库和装饰器这样的设计模式，我们可以看看这个框架的实现方式。从思路上看，整个框架需要完成以下两个主要工作。

（1）在定义时传入一些设定好的共享参数，并将其进行保存，必要时还可以实现参数的嵌套覆盖等机制，这部分工作主要由 contextlib 完成。

（2）在函数被调用时，将设定的默认参数传入函数中。这部分工作主要由装饰器完成。

接下来，我们就来介绍这两个工作。这部分的代码在 tensorflow/contrib/framework/python/ops/arg_scope.py 中。

1. contextlib 部分

contextlib 是 Python 提供的帮助用户管理环境信息的库，我们可以通过它来实现一些控制部分代码块的功能。假设我们想实现一个基于代码块的计时器，可以这样定义：

```
1 from contextlib import contextmanager
2 import time
3
4 @contextmanager
5 def timer(prefix):
6     start_time = time.time()
7     yield()
8     duration = time.time() - start_time
9     print prefix + str(duration)
```

代码的第 7 行是一个 yield() 函数，执行这个函数时，程序将暂时离开这个 context 函数，继续执行 context 内我们定义的代码，等到 context 内的代码执行完毕，再回到这个函数，继续执行下面的第 8 行和第 9 行代码。定义完这个计时器后，我们只要在对应的代码块上调用它即可：

```
with timer('this block'):
    # 你的代码
```

仅供非商业用途或交流学习使用

通过这种方法，我们以相对简洁的方式在代码块内插入了一段代码，同时不会造成代码混乱。利用这个方法，我们可以创建一个上下文对象（Context Object），用于存储要共享的参数。这部分参数将被保存在文件的_ARGSTACK 变量中，这是一个全局变量。考虑到嵌套 arg_scope 的需要，变量采用栈结构实现，内层的 arg_scope 的优先级将高于外层的 arg_scope。

每当一个 arg_scope 被声明出来，最近保存的共享参数集合就会被取出来，新的参数将会和旧的参数融合。如果有相同的参数，那么新的参数将覆盖旧的参数。这样一份新的共享参数集合就诞生了，我们将其保存到_ARGSTACK 的栈顶，如果这个 context 结束，就把这份参数集合从栈中弹出删除。

这部分的核心代码如下：

```
@tf_contextlib.contextmanager
def arg_scope(list_ops_or_scope, **kwargs):
    try:
        #将上一个共享参数集合取出
        current_scope = _current_arg_scope().copy()
        # 对我们要设置的每一个函数进行参数更新
        for op in list_ops_or_scope:
            key_op = _key_op(op)
            # 如果参数在上一个集合已经存在，则替换
            if key_op in current_scope:
                current_kwargs = current_scope[key_op].copy()
                current_kwargs.update(kwargs)
                current_scope[key_op] = current_kwargs
            else: # 如果不存在，则添加
                current_scope[key_op] = kwargs.copy()
        # 将新的集合推入参数栈中
        _get_arg_stack().append(current_scope)
        yield current_scope
    finally:
        _get_arg_stack().pop()
```

通过使用 contextlib，我们已经完成了默认参数的存储工作。虽然参数已经保存好，但是我们仍然希望函数在调用时能够利用这些参数，为此我们需要另一部分功能的帮助。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2. 装饰器部分

装饰器也是 Python 中的一个功能，每个函数通过装饰器的封装，可以在不改变函数代码的情况下为它增加功能。例如，同样是实现一个拥有计时功能的装饰器：

```
import time
def deco(func):
    start_time = time.time()
    func()
    duration = time.time() - start_time
    print prefix + str(duration)
    return func
```

定义完成后，只要在函数前面加上与上面定义的函数名同名的注解（Annotation）即可，代码如下：

```
@deco
def my_func():
    # 你的代码
```

这样在执行时，就可以看到程序被执行的效果，并保证代码的整洁和可读性。

看到这里，读者可能会想，上面介绍的两个功能都是为既有的代码注入一些新功能的代码，这两者有什么区别呢？contextlib 是为一段代码的环境注入代码，而装饰器是为一个函数注入代码，它们注入的目标不同。contextlib 的功能是保存默认参数，所以装饰器的功能就是把注入的参数传入制定的函数中。

所以每一个被 arg_scope 包围的函数，如果想实现自动传入默认参数的功能，都需要一个装饰器辅助，装饰器可以确保函数在执行前先执行装饰器的代码。装饰器会在调用函数前从_ARGSTACK 中把参数取出来，和函数本身的参数进行融合，再一同放入调用的函数中，这样就完成了参数共享的工作。对应的代码如下所示：

```
def add_arg_scope(func):
    def func_with_args(*args, **kwargs):
        current_scope = _current_arg_scope() # 取出当前的共享参数集合
        current_args = kwargs
        key_func = _key_op(func)
        #如果当前的函数在集合中，加入共享参数if key_func in current_scope:
            current_args = current_scope[key_func].copy()
            current_args.update(kwargs)
        return func(*args, **current_args) # 完成真正的调用
```


書籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    _add_op(func) # 将当前函数存入可装饰的函数集合中
    setattr(func_with_args, '_key_op', _key_op(func))
    return tf_decorator.make_decorator(func, func_with_args)
```

在使用时，需要对 TensorFlow 原生的函数做一定的封装，这样就可以直接在 arg_scope 中使用它了。我们可以使用一个小例子展示它的效果。具体的代码如下所示：

```
@add_arg_scope
def func1(*args, **kwargs):
    return (args, kwargs)

with arg_scope((func1,), a=1, b=None, c=[1]):
    args, kwargs = func1(0)
    print args
    print kwargs
```

上面这段代码的输出结果为：

```
(0,)
{'a': 1, 'c': [1], 'b': None}
```

可以看到 arg_scope 中设定的参数被传入了函数中。以上就是 arg_scope 的原理及应用，相信这个功能能够帮助读者节省不少写代码的时间。

至此，我们完成了对 TensorFlow 部分实现原理的介绍，实际上本章介绍的只是 TensorFlow 的冰山一角，真正复杂的部分并未涉及，对原理强烈好奇的读者可以深入学习。

4.3 TensorFlow 的分布式训练

本节介绍使用 TensorFlow 进行分布式训练的方法。虽然 GPU 及其他专用处理器的出现使计算效率有了惊人的提升，但我们对模型的野心是不断膨胀的。面对不断增长的数据，越来越复杂的模型，单机单 GPU 训练已经变得力不从心，单机多 GPU 训练也显得有些乏力，于是我们开始追求更高的目标：分布式训练。分布式训练更有利于运算资源扩展，通过若干台机器的协作，计算效率可以达到接近线性效果的增长，这极大地减少了运算时间。

图 4-8 所示为分布式同步训练的整体流程。首先，分布在不同机器上的进程（process）完成了必要的通信，彼此确认了对方的存在；然后，所有机器完成参数初始化的

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

同步，所有进程拥有相同的参数；此后在每一轮训练过程中，每一个进程分别完成自己的前向计算（forward）和后向计算（backward），并得到每一个参数的更新量；接着所有的进程进行同步（All-Reduce），对所有的参数更新量求平均，得到基于所有进程训练数据的平均参数更新量；最后，每个进程将获得的更新量应用到自己的参数上，此时每一个进程的参数享受所有进程的计算，同时保持了参数的一致，然后进行下一轮的训练计算。

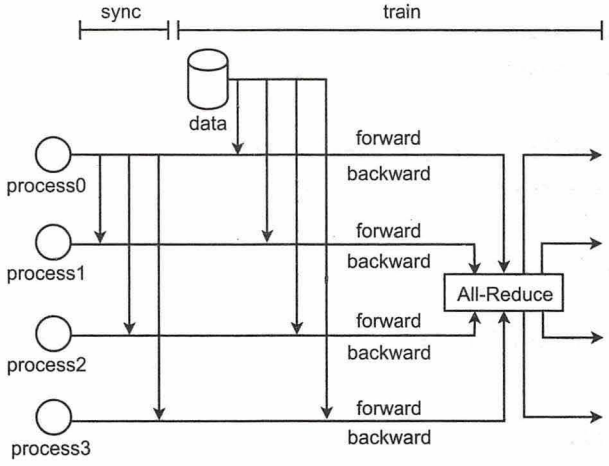


图 4-8　分布式训练的示意图

从计算流程来看，分布式计算和单机多 GPU 计算没有太大的区别，主要的区别在于参与同步的数据没有在同一台机器上。原本我们在一台机器上只有 4 个或者 8 个 GPU，现在通过通信设施，可以将更多 GPU 组织在一起。

在论文 *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour* 中，作者介绍了团队是如何利用分布式集群完成模型的快速学习的。除了介绍很多训练过程中应该注意的技巧外，作者还介绍了自己使用的分布式架构。文章中包含很多有实用价值的技巧，因这部分知识和本书的关系并不密切，故不展开，对这些内容感兴趣的读者可以自行阅读。

接下来介绍如何实现分布式训练模型。在此之前，笔者先介绍分布式同步训练的关键技术之一：MPI 协议及其操作。

4.3.1　基于 MPI 的数据并行模型

MPI 的全称是 Message Passing Interface，它是一套十分经典的解决多进程通信问题的框架，框架中定义了一些最基本的多进程通信协议，并定义了这些协议的详细参数

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

和调用方式，使我们在一些问题上可以直接使用这套框架解决问题。

前面我们提到，分布式同步训练需要不同进程间相互通信，这里涉及很多通信的细节：要用什么样的方式进行通信？TCP、InfiniBand，还是其他？采用包含中心节点的架构进行通信，还是采用无中心的方法？节点间的关系如何表达？进程出现错误该怎么办？这些琐碎的问题包含了很多技术细节，这些细节很可能不是一个研究机器学习的人擅长的。于是有前辈提出了一套方案：用统一的接口定义进程间通信的行为，至于通信的细节则由专业人士开发完成，使用者无须精通。MPI 就是这样的一套接口。

在使用 MPI 的接口时，我们需要遵循一定的基本定义。使用 MPI 框架时，所有进程将运行相同的程序。程序开始时，所有进程将通过通信组成一个 Group，Group 内的进程彼此知晓对方，能够协同工作。每一个进程可以得到两个用于协同的数值。

（1）size：表示 Group 内总的进程数。

（2）rank：表示自己在 Group 中的序号。MPI 的实现会为 Group 中的每一个进程提供唯一的 rank 值。

作为应用的开发者，我们可以通过 rank 为每一个进程安排不同的工作，从而完成通信与协作。MPI 最基础的操作就是 send 和 receive。

（1）send：把数据从一个进程发送到另一个进程。

（2）receive：等待接收其他进程发来的数据。

基于上面提到的两个基本操作，我们实现了 MPI 中的一些经典操作，下面就来看看这些操作。

1. Scatter

Scatter 操作用于将一个进程的数据发送到同一环境的所有进程中。数据发送的过程并不是将一份数据同时复制到多台机器中，而是将数据切成 N 份，每一个进程收到一份数据。

我们假设有 4 个进程：A、B、C 和 D。数据发起者为进程 A，接收者为 A、B、C 和 D。启动前，所有的数据都在进程 A，因此待传输的数据被分成 1、2、3、4 共 4 份，保存在进程 A 中，操作完成后，4 份数据将按顺序分别传输到 4 个进程中。操作的具体流程如图 4-9 所示。

2. Gather

Gather 操作用于将多个进程的数据聚集到单一进程中。Gather 操作可以理解为 Scatter 操作的逆操作。假设有 4 个进程：A、B、C 和 D。数据发起者为进程 A、B、C 和

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

D，接收者为进程 A。启动前每个进程拥有 1 份数据，操作过程中每个进程都将自己的数据发送到进程 A，操作完成后，4 份数据将按照进程的顺序放置到进程 A 中。操作的具体流程如图 4-10 所示。

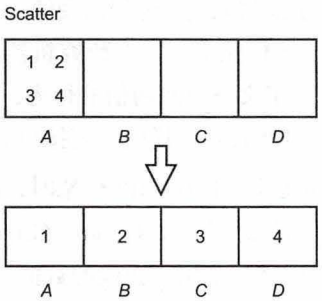


图 4-9　Scatter 操作的示意图

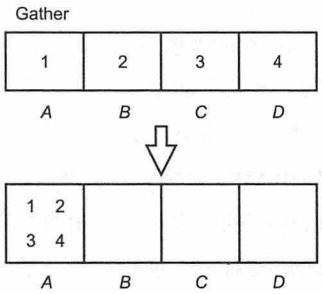


图 4-10　Gather 操作的示意图

3. Allgather

Allgather 操作将 gather 应用到所有的进程，gather 操作中只有一个进程收到所有进程发来的数据，而对于 Allgather 操作，所有的进程都会收到这些数据，操作流程如图 4-11 所示。

4. MPI_Bcast

这个操作名是广播（Broadcast）的缩写，用于将一个进程的数据发送给所有的进程。与 Scatter 不同的是，Bcast 并不将数据分成 N 份，而是将所有的数据发出，操作流程如图 4-12 所示。

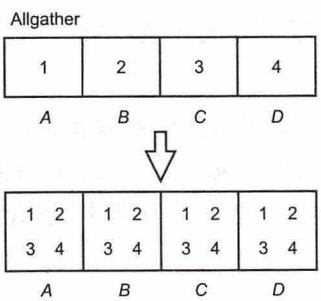


图 4-11　Allgather 操作的示意图

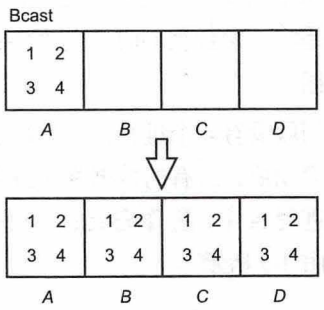


图 4-12　Bcast 操作的示意图

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

5. MPI_Reduce

这个操作类似 Gather，将所有进程的数据汇聚到单一进程中。与 Gather 不同的是，Reduce 操作将对这些数据进行某种操作，例如求和、求最大值等，操作流程如图 4-13 所示。

6. MPI_Allreduce

这个操作从名字来看显然和 Reduce 有关。就像 Gather 和 Allgather 的关系，Reduce 和 All-Reduce 也只是在接收数据的数量上有所区别。这个操作是分布式同步优化最关键的操作，操作流程如图 4-14 所示。

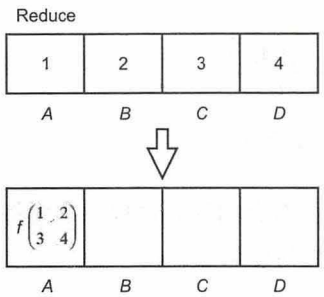


图 4-13 Reduce 操作的示意图

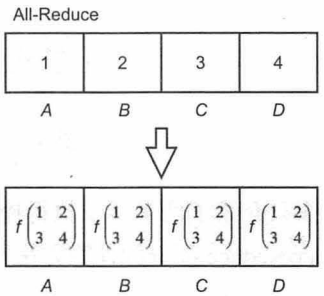


图 4-14 All-Reduce 操作的示意图

看完了上面对这些经典的通信模型的介绍，读者一定会想，这些操作能用到什么地方？它们是如何实现的呢？笔者先来回答第一个问题。在分布式同步训练中，我们主要用到两个操作：MPI_Bcast 和 MPI_Allreduce。MPI_Bcast 用于同步所有进程的初始化参数，由于每一个进程都可以对参数进行初始化，它们对参数的初始化效果是可以等价的，所以可以选择其中一个进程作为发送方，将其参数值发送到其他进程上，这样参数的初始化同步就完成了。MPI_Allreduce 用在参数更新量的求和上，我们将所有参数聚集起来，并对其进行求和，最后将这些结果发送到每一个进程上。

第二个问题涉及对 MPI 算法细节的了解。笔者主要介绍在训练中出现频率最高的 All-Reduce 方法。从它的工作原理中我们可以看出整个流程涉及数据的聚集和分发，其中聚集部分一般被称为 Scatter-Reduce，也就是说每一个进程会完成部分数据的聚集和计算；而分发部分就是 All-Gather 操作，将每一个小部分的计算结果发送到所有的进程中，详细过程如图 4-15 所示。

从图 4-15 可以看出，在计算前每一个进程会根据 Group 的数量将待通信数据划分成相同数量的部分，如果 Group 内有 4 个进程，那么每个进程都要把数据切成 4 份。由

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

于计算时每一个位置数据的计算是独立的，因此切分后的数据可以独立计算。为什么要采用这样的计算流程，把计算分散到每一个进程，而不是把所有的数据聚集到一个进程中呢？把所有的数据聚集到一个进程中当然是可以的，但会导致每个进程的计算量不同，有些机器（例如聚集的机器）耗时较长，有些机器则处于闲置。为此我们需要把工作量尽可能地分摊到所有的机器上。

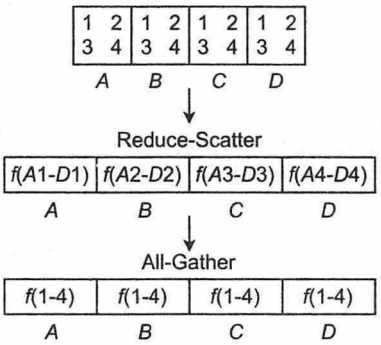


图 4-15　All-Reduce 的计算流程

上面只介绍了两个步骤执行操作后的结果，并没有介绍计算过程，下面将重点介绍 Scatter-Reduce 部分的计算方法，All-Gather 的计算与此类似。

第一种方法被称为 Ring Method。之所以叫这样的名字，是因为方法将所有进程的通信的拓扑结构想象成环状，如果 Group 中有 4 个进程，分别被称为 A、B、C 和 D，每个进程的数据将分成 4 份，分别为 Segment1、Segment2、Segment3 和 Segment4。它们的组织结构如图 4-16 所示。

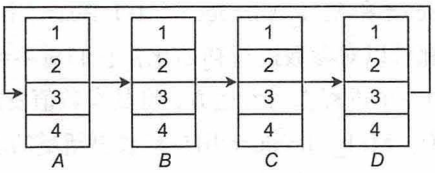


图 4-16　Ring Method 的进程拓扑结构

从图 4-16 中可以看出，基于这样的拓扑结构，对于进程 i 来说，它只能和进程 $i+1$ 和进程 $i-1$ 进行通信，一般来说它将数据发送到进程 $i+1$，并接收来自进程 $i-1$ 的数据。从数据片段的角度看，每个片段需要穿越这条环线将数据聚集，因此整个过程将持续 $N-1$ 个循环。对有 4 个进程进行通信的场景，在第一轮循环中，每个数据片段的传输流程如下。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

Segment0：进程 0→ 进程 1

Segment1：进程 1→ 进程 2

Segment2：进程 2→ 进程 3

Segment3：进程 3→ 进程 0

在第二轮循环中，每个数据片段的传输流程如下。

Segment0：进程 1→ 进程 2

Segment1：进程 2→ 进程 3

Segment2：进程 3→ 进程 0

Segment3：进程 0→ 进程 1

最后一轮循环中，每个片段的数据再遵循类似的方式传输数据，就可以完成通信的全过程，以上过程如图 4-17 所示。

Ring Method 的时间和进程数量有关。进程越多，时间越长，这是 Ring Method 方法的短板，但它的优点是对 Group 的组织形式没有特别要求，数据传输时对带宽压力也相对较小。

第二种方法被称为 Recursive Halving & Doubling Method。这个算法在循环轮数上比 Ring Method 少，但是要求进程的总数为 2 的 N 次方，这样才能获得最好的效果。在这个方法中，进程的通信拓扑结构也有所不同，我们要求每一个进程和 N 个进程进行通信，同时对进程的通信量需求也更大。对应的拓扑结构如图 4-18 所示。

这个方法共执行 N 轮循环，每一轮循环的数据通信方式都不相同。我们仍以四个进程 A、B、C、D 为例，看看它的通信方式。在第一轮循环中，相邻的两个进程互相交换数据，进程 0 和进程 1 互换，进程 2 和进程 3 互换，每一个进程都将自己一半的数据发送给对方，同时收集对方一半的数据；在第二轮循环中，距离为 2 的两个进程相互交换数据，进程 0 和进程 2 互换，进程 1 和进程 3 互换，每一个进程都将自己四分之一的数据发送给对方，同时收集对方一半的数据，整个流程如图 4-19 所示。

在 Recursive Halving & Doubling Method 中，由于前面几轮数据传输量比较大，系统的整体带宽存在很大的压力。同时，它对系统内的进程数量有要求，不满足指定数量的线程时，需要对算法进行一定的改变才能使用。虽然这个方法减少了循环次数，但是在实际执行中与不同的进程通信和传输大量数据也会造成一定的时间损耗，因此理论上 Recursive Halving & Doubling Method 会更快一些，但是实际上有时 Ring Method 执行得更快。读者需要根据自己的硬件环境情况选择合适的通信方法。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

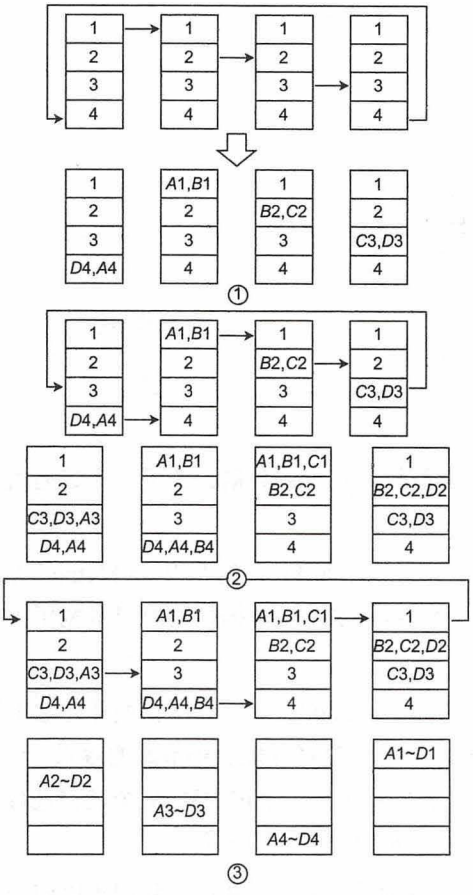


图 4-17　Ring Method 的执行流程

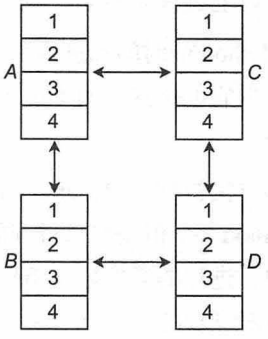


图 4-18　Recursive Halving & Doubling Method 的拓扑结构

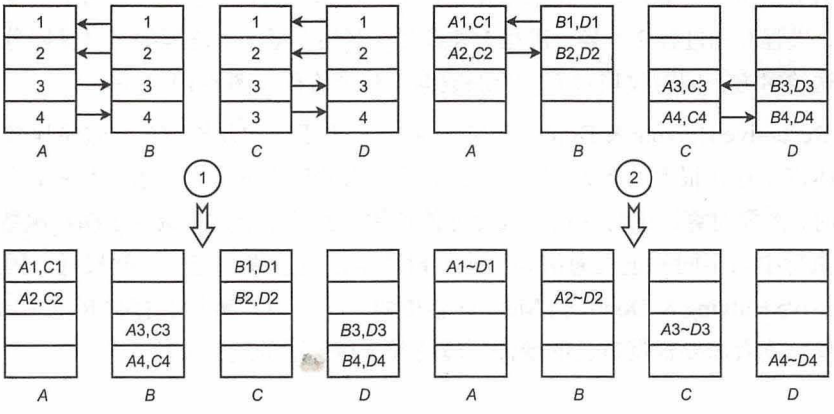


图 4-19　Recursive Halving & Doubling Method 的执行流程

仅供非商业用途或交流学习使用

4.3.2 MPI 的实现：mpi_adam

下面我们来看 MPI 在实际代码中的实现，我们从在第 5 章将介绍的 Baselines 代码中可以找到它的实现。代码在 common/mpi_adam.py 中，它的核心代码如下：

```
def update(self, localg, stepsize):
    if self.t % 100 == 0:
        # 检查参数是否一致
        self.check_synced()
    localg = localg.astype('float32')
    globalg = np.zeros_like(localg)
    self.comm.Allreduce(localg, globalg, op=MPI.SUM)
    if self.scale_grad_by_procs:
        globalg /= self.comm.Get_size()

    self.t += 1
    step = ... # 利用globalg计算更新量
    self.setfromflat(self.getflat() + step)
```

从代码中可以看出，在更新参数时，每隔一段时间，系统会检查所有进程的参数是否一致，再进行 All-Reduce 操作，最后除以进程数量，把更新量赋给参数。这和我们前面提到的流程完全一致。在执行时我们可以使用这样的命令：

```
mpirun -np 4 -H [ip1]:2,[ip2]:2 [command]
```

在这个命令中，我们使用 MPI 执行程序，这个 Group 中有 4 个进程，其中 ip1 的机器中有 2 个进程，ip2 的机器中有 2 个进程。它们的通信方式可以通过其他参数进行设定。

当然，这个方法也存在一定的缺陷，我们需要把本地所有参数的梯度计算出来，再进行 All-Reduce 操作。这个过程不会造成结果的偏差，但会造成时间的浪费。因为计算完梯度，就相当于进行了一次同步操作，这个计算过程如图 4-20 所示。

实际上，中间一段梯度的同步工作可以省略，任意一个参数的梯度计算完成后，就可以进行这个参数的同步，所以更好的更新流程如图 4-21 所示。

至于这种一段式的更新流程，请读者自行思考实现完成，在此不再赘述。

仅供非商业用途或交流学习使用

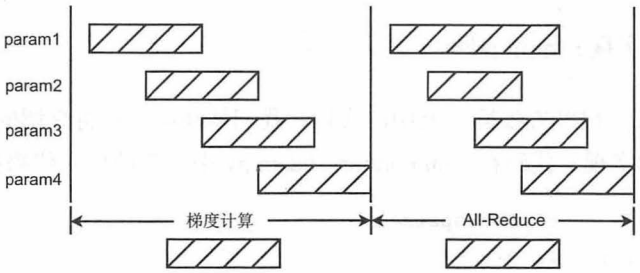


图 4-20　两段式梯度更新流程

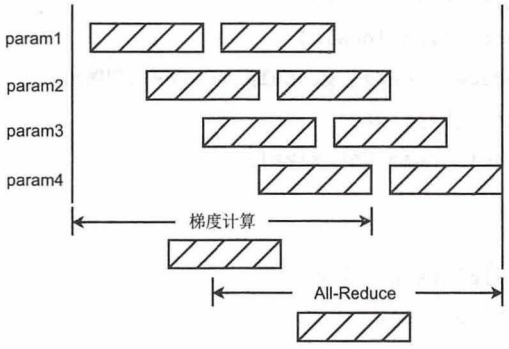


图 4-21　一段式梯度更新流程

4.4　基于 TensorFlow 实现经典网络结构

为了让读者更快地掌握 TensorFlow 的使用方法，本节将展示基于 TensorFlow 的一些经典网络实现，这些网络包括多层感知器（Multi Layer Perceptron，MLP）、卷积神经网络（Convolutional Neural Network，CNN）和循环神经网络（Recurrent Neural Network，RNN）。

4.4.1　多层感知器

多层感知器主要由全连接层和非线性激活层组成，全连接层一般由矩阵乘法表示，输入的向量 $\boldsymbol{x}$ 和参数矩阵 $\boldsymbol{W}$ 相乘后，得到的结果和向量 $\boldsymbol{b}$ 相加，就得到了结果 $\boldsymbol{z}$，计算过程如下所示：

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

仅供非商业用途或交流学习使用

由于输出中的每一个值要经过所有输入值的计算，因此这个结构被称为全连接层。非线性激活层一般由线性整流函数（Rectified Linear Unit，ReLU）表示，它可以将输入的负数部分变为 0，这样模型便满足了一定的非线性效果。对于一个 3 层网络组成的感知器，它的结构如图 4-22 所示。

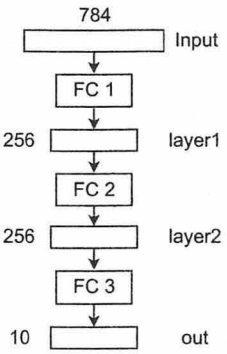


图 4-22　3 层感知器的结构图

对应的 TensorFlow 代码如下所示：

```
# 网络参数
n_hidden_1 = 256 # 第1层网络神经元数量
n_hidden_2 = 256 # 第2层网络神经元数量
n_input = 784 # 数据输入维度
n_classes = 10 # 数据输出维度

# placeholder
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# 模型中的参数定义
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
```

仅供非商业用途或交流学习使用


```
}

def multilayer_perceptron(x):
    # 第一层全连接层和非线性层，下同
    layer_1=tf.nn.relu(tf.add(tf.matmul(x, weights['h1']), biases['b1']))
    layer_2=tf.nn.relu(tf.add(tf.matmul(layer_1, weights['h2']), biases
        ['b2']))
    out = tf.matmul(layer_2, weights['out']) + biases['out']
    return out
```

4.4.2 卷积神经网络

随着近年来的发展，卷积神经网络已经成功应用于很多视觉、语音、自然语言处理的任务上。卷积神经网络的卷积层可以在提取局部信息的同时尽可能地减少计算量。VGG Net 是卷积神经网络曾经的代表作，它在网络结构设计上拥有自己的特点，其中一些设计思路也被很多网络沿用至今，它的网络结构如图 4-23 所示。

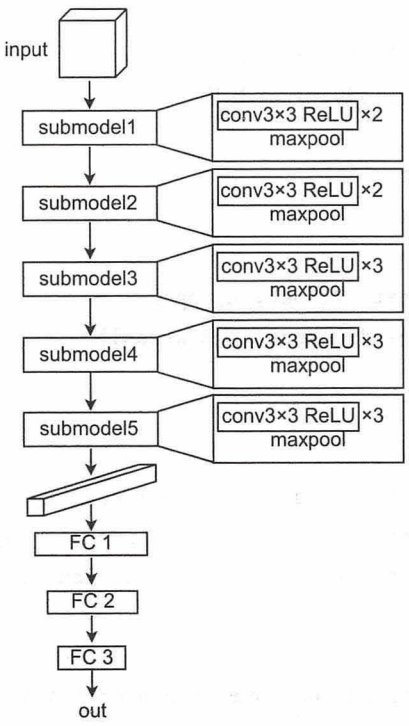


图 4-23　卷积神经网络结构图

仅供非商业用途或交流学习使用

从图 4-23 中可以看出，整个模型分为 6 个部分，前 5 个部分由卷积层组成，最后一个部分由全连接层组成。在每一个卷积部分内，图像的尺度并没有发生变化，只有在进入下一个部分时，图像的长宽依次减半，但与此同时，它的通道数也扩大了一倍，这就相当于图像的信息在不断地集中变少，而对图像分析的方式在不断增多。VGG Net 的很多设计也被后来的模型使用。

对应的实现代码如下：

```
# 5个卷积部分的输出通道数
filter_depth = [64, 128, 256, 512]
VGG16_NUM_HIDDEN_1, VGG16_NUM_HIDDEN_2 = 4096, 1000

def submodel(x, iters, k_width, in_depth, out_depth):
    for i in range(iters):
        cur_in_depth = in_depth if i == 0 else out_depth
        w = tf.Variable(tf.truncated_normal([k_width, k_width,
        cur_in_depth, out_depth], stddev=0.01))
        b = tf.Variable(tf.zeros([out_depth]))
        x = tf.nn.relu(tf.nn.conv2d(x, w, [1,1,1,1], padding='SAME') + b)
    x = tf.nn.max_pool(x, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')
    return x

def fc(x, in_size, out_size):
    w = tf.Variable(tf.truncated_normal([in_size, out_size]))
    b = tf.Variable(tf.constant(0.0, shape=[out_size]))
    out = tf.matmul(x, w) + b
    return tf.nn.relu(out)

def vgg16(x, filter_depth, hidden1, hidden2, img_height, img_width,
img_depth, num_labels):
    out = x
    k_w = 3
    # 1st part
    k_d1 = filter_depth[0]
    out = submodel(out, 2, k_w, img_depth, k_d1)
    # 2nd part
    k_d2 = filter_depth[1]
    out = submodel(out, 2, k_w, k_d1, k_d2)
```

仅供非商业用途或交流学习使用


```
    # 3rd part
    k_d3 = filter_depth[2]
    out = submodel(out, 3, k_w, k_d2, k_d3)
    # 4th part
    k_d4 = filter_depth[3]
    out = submodel(out, 3, k_w, k_d3, k_d4)
    # 5th part
    out = submodel(out, 3, k_w, k_d4, k_d4)

    element_num = ((img_height // (2 ** 5)) * (img_width // (2 ** 5)) *
        k_d4)
    out = tf.reshape(out, [-1, element_num])

    # fully connected part
    out = fc(out, element_num, VGG16_NUM_HIDDEN_1)
    out = tf.nn.dropout(out, 0.5)
    out = fc(out, VGG16_NUM_HIDDEN_1, VGG16_NUM_HIDDEN_2)
    out = tf.nn.dropout(out, 0.5)

    return fc(out, VGG16_NUM_HIDDEN_2, num_labels)
```

卷积神经网络的世界还有很多优秀的模型结构，例如残差网络等，希望读者能够自行学习了解。

4.4.3 循环神经网络

在前面介绍的模型中，我们可以通过传入某个固定的输入得到固定的输出，模型并不能对输入数据产生记忆。当我们遇到一些序列预测问题时，这种基于时间或位置关系的问题又需要考虑时间或空间上的依赖关系，因此前面提到的模型不能满足这样的需求，于是我们就需要使用循环神经网络。循环神经网络实现了对序列数据的建模和拟合，当我们使用模型计算出某一输入数据的结构后，部分信息就被记录到模型中，这些信息会在下一个输入计算时产生作用，这些记录的信息被称为隐含层的状态信息。模型的整体结构如图 4-24 所示，其中 $\boldsymbol{x}$ 表示模型的输入，$\boldsymbol{h}$ 表示模型的隐含状态，$\boldsymbol{o}$ 表示每一时刻的输出，而 $\boldsymbol{U}$、$\boldsymbol{V}$ 和 $\boldsymbol{W}$ 表示模型的参数。图 4-24 中左边展示的是聚合版本的模型，经过展开后可以得到右边的样子，这样可以清晰地看出时刻之间的依赖关系。

仅供非商业用途或交流学习使用

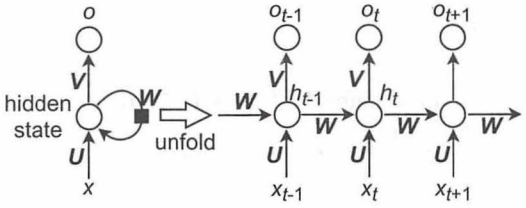


图 4-24 循环神经网络原理图

经典的 RNN 模型在训练时存在梯度消失的问题，为了解决这个问题，前人发明了很多改进版的模型，而 LSTM（Long Short Term Memory）是其中最常见的模型，它能很好地解决因为时间跨度长而导致的梯度消失问题，它的基本结构如图 4-25 所示。

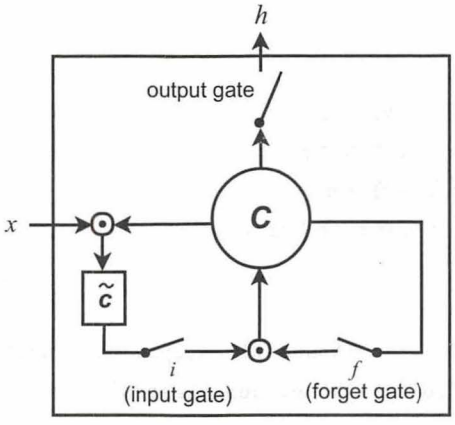


图 4-25 LSTM 的结构图

图 4-25 中的变量分成以下几类。

- 输入：$\boldsymbol{x}$。
- 模型内部的状态：$\boldsymbol{c}$ 表示 cell 的状态值，$\boldsymbol{h}$ 表示隐层的状态值。
- 模型内部的三道门及对应的参数变量：输入门 $\boldsymbol{i}$、$\boldsymbol{W}^{(i)}$ 和 $\boldsymbol{U}^{(i)}$；遗忘门 $\boldsymbol{f}$、$\boldsymbol{W}^{(f)}$ 和 $\boldsymbol{U}^{(f)}$；输出门 $\boldsymbol{o}$、$\boldsymbol{W}^{(o)}$ 和 $\boldsymbol{U}^{(o)}$。

这里的计算过程分为三步，第一步计算三个门的值，三个门用于控制各部分信息参与运算的量，对控制记忆和保证梯度传递起到重要的作用：

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}^{(i)}\boldsymbol{x}_t + \boldsymbol{U}^{(i)}\boldsymbol{h}_{t-1})$$
$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}^{(f)}\boldsymbol{x}_t + \boldsymbol{U}^{(f)}\boldsymbol{h}_{t-1})$$
$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}^{(o)}\boldsymbol{x}_t + \boldsymbol{U}^{(o)}\boldsymbol{h}_{t-1})$$

第二步计算 cell 的值：

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{W}^{(h)}\boldsymbol{x}_t + \boldsymbol{U}^{(h)}\boldsymbol{h}_{t-1})$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$$

第三步计算隐层的状态值，也是模型的输出值：

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$$

由于这部分的计算比较复杂，TensorFlow 对其进行了封装，我们只需要简单地定义每一步使用的 RNN 结构和模型的长度等基本信息，就可以构建出完整的计算结构。对应的代码如下所示：

```
# 网络参数
num_input = 28 # 数据输入的维度
timesteps = 28 # 数据输入的总时长
num_hidden = 128 # 隐含层的维度
num_classes = 10 # 算法最终的输出维度

# placeholder
X = tf.placeholder("float", [None, timesteps, num_input])
Y = tf.placeholder("float", [None, num_classes])

# 定义LSTM之外的全连接层的参数
weights = {
    'out': tf.Variable(tf.random_normal([num_hidden, num_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([num_classes]))
}

def RNN(x, weights, biases):
    x = tf.unstack(x, timesteps, 1)
    lstm_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0)
    outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)

    return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

这部分代码实现了一个根据序列进行单一结果预测的模型，我们只需要最后一个时刻的输出信息，并将其传入全连接层进行处理，得到最终结果，模型对应的原理图如图 4-26 所示。

读者了解了上面这些网络的实现方式，就能较轻松地完成网络构建相关的操作。

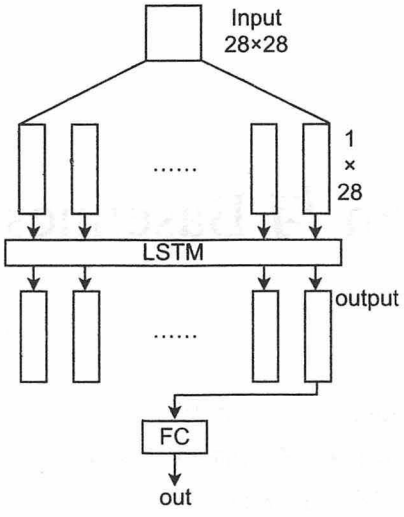


图 4-26　示例代码对应的模型图

4.5　总结

本章介绍了 TensorFlow 的基本使用方法和一些常见的与训练相关的内容，让我们回顾一下。

（1）TensorFlow 的使用主要包括：计算图定义、操作运算地点安排和计算图执行三部分。

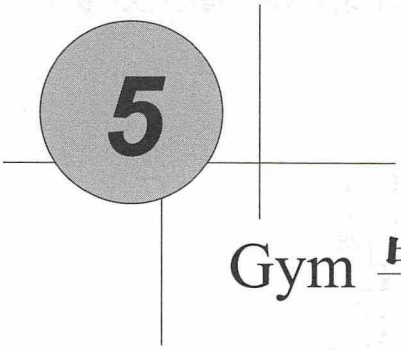
（2）TensorFlow 内部的一些运行机理很值得读者做一定的研究。

（3）使用 MPI 的通信方式可以使 TensorFlow 完成分布式训练。

（4）TensorFlow 可以实现常见的各种经典模型。

4.6　参考资料

[1] https://www.tensorflow.org/?hl=zh-cn



Gym 与 Baselines

第 4 章介绍了 TensorFlow 这个十分强大的工具，它可以帮助我们很好地完成机器学习相关的工作。但是对本书主要介绍的强化学习来说，它还欠缺很多。为此，我们还需要寻找其他工具辅助完成 TensorFlow 没有完成的工作，于是我们找到了 Gym 和 Baselines 两个优秀的项目。Gym 主要用于生成常见的强化学习模拟环境，有了它我们就可以直接进行交互学习，不必考虑环境中的种种复杂逻辑。Baselines 则是一个包含了很多经典强化学习算法的项目，我们可以通过直接使用项目中的代码来感受算法的效果，也可以通过阅读代码了解算法的原理。本书也将介绍项目中大部分的算法。

5.1 Gym

在本章开始时我们已经介绍了 Gym 的作用：生成常见的强化学习模拟环境。为了快速地完成学习，一个简单方便的环境模拟器非常重要。在 Gym 之前，已经有很多优秀的仿真平台被创建，它们也具备不错的易用性，但是 Gym 具备了自己的特点：使用方便，同时集成了大量的仿真环境。本节就来介绍 Gym 的基本使用方法。

5.1.1 Gym 的安装

Gym 的安装过程并不复杂，执行以下命令就可以完成安装：

```
git clone https://github.com/openai/gym.git
```

```
cd gym
pip install -e .
```

或者执行：

```
pip install gym
```

完成这一步后，我们只完成了最基础的 Gym 库安装。如果想要将对应的游戏环境安装好，还需要在 Gym 的代码目录下执行以下命令：

```
# 安装atari环境
pip install -e '.[atari]'
# 安装棋盘类游戏
pip install -e '.[board_game]'
# 安装Box2d控制类游戏
pip install -e '.[box2d]'
# 安装经典控制类游戏
pip install -e '.[classic_control]'
# 安装MuJoCo控制类游戏
pip install -e '.[mujoco]'
```

上面的命令每执行一句，系统就会安装一类游戏。以 Atari 为例，里面有我们熟悉的空间入侵者（SpaceInvaders）、吃豆人（MsPacman）等经典游戏，也有考验模型探索能力的蒙特祖玛的复仇（Montezuma's Revenge），如图 5-1～图 5-3 所示。

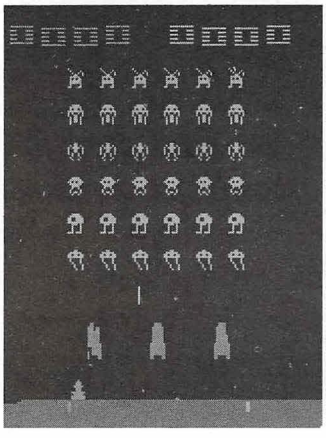


图 5-1　空间入侵者游戏

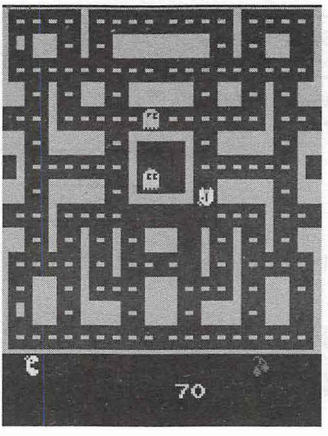


图 5-2　吃豆人游戏

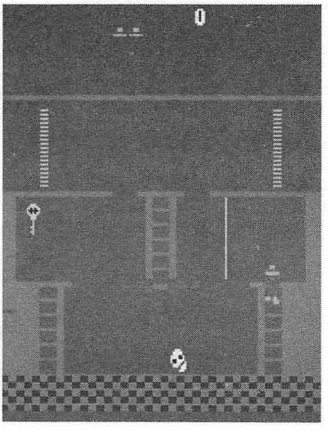


图 5-3　蒙特祖玛的复仇

5.1.2 Gym 的基本使用方法

Gym 官方给出了它的基本使用方法，还提供了一个最简单但十分“通用”的 Agent。在此，笔者将对 Gym 的基本使用方法和它的架构进行更详细的介绍。

Gym 中接触的第一个对象就是**环境（简称 Env）**。它和强化学习概念中的环境类似，模拟了我们交互的环境，同时将一些经常用到的功能封装起来。我们只需要知道游戏的名字就可以创建对应的环境：

```
import gym
env=gym.make('[game name]')
```

env 有如下三个核心方法。

（1）reset：重新开始任务，将环境设置成任务初始的状态 s_0。

（2）step(a)：这个方法要传入一个参数 a，这个参数代表 Agent 在此时刻将要完成的行动。环境会依照行动模拟生成一系列的信息：下一时刻的观测状态、获得的奖励、此次任务是否已经结束，以及其他信息。

（3）render：输出当前的状态。5.1.1 节中的图片都是通过这个函数生成的。

加上 Agent 的操作，就可以将其组成一个完整的强化学习的交互流程，如图 5-4 所示。

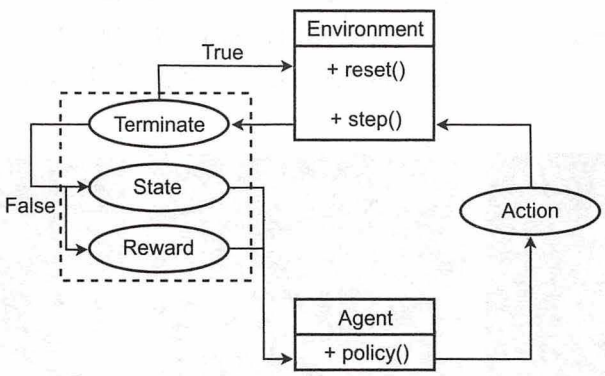


图 5-4 基于 Gym 的强化学习交互流程

了解了这三个方法，就可以使用它们实现一个最简单的 Agent：“佛系”Agent。这个 Agent 的行动完全随机，它不考虑每一步行动可能带来的影响。基本代码如下所示：

```
1 import gym
2 import sys
```

```
3
4  env = gym.make(sys.argv[1])
5  env.reset()
6  for i in range(1000):
7      env.render()
8      ob, reward, done, info = env.step(env.action_space.sample())
9      if done:
10         env.reset()
```

代码的第 8 行就是这个 Agent 执行的关键：它从 env 中获得了问题的行动空间，然后从中进行采样，得到一个随机行动。实际上，本书的主要任务就是将这行代码换成一个更强大的实现，这需要将随机操作换成更强大的模型和训练方法。

1. Gym 对空间的定义

实际上，Gym 并没有实现太多环境细节，只是将很多的环境集成在自己的框架下。为了统一每一个平台的环境接口，Gym 实现了一套关于 MDP 的基本框架，并将其他平台的接口适配到这套框架下。对每一个任务，我们都需要描述状态所处的空间和行动所处的空间。定义好这两个空间，模型接口的数据传输维度就基本确定了。为了简化操作，Gym 中实现了一些常见的空间供用户使用。

Box：Box 用于实现连续数据构成的空间，其中包含两组参数：空间内数据的范围（上限和下限），以及空间维度的大小，我们可以这样定义空间的形式：

```
Box(-1.0, 1.0, (3,4))
```

这样就定义了一个数值在 $-1 \sim 1$ 之间，维度为 3×4 的空间。这个类型的空间主要使用在 Atari 游戏和棋盘类游戏状态中。Atari 的状态主要是二维的游戏画面，棋盘也是一个二维的数据，这两个空间都很适合用 Box 表示。

Discrete：这个空间描述了一个 Categorical 分布的空间，空间由 N 个离散状态构成，每一个状态之间相互排斥，我们只能从中选择一个状态。它一般被用在一些分类问题上，且主要用于描述行动所在的空间。它的定义形式为：

```
Discrete(2)
```

这其实就是一个 Bernoulli 分布所在的空间。

2. Wrapper

前面介绍 Gym 提供了环境最基本的功能，如显示基本的环境，根据行动进入下一个状态等。在现实中，我们有时需要对环境做一些改变，扩展新的功能，这就需要对环境代码做改变。为了更方便地扩展，Gym 中定义了 Wrapper 这个类，它可以在既有环境的基础上添加更多的功能。这个类在创建时需要传入一个 env 对象，它是一个已经创建好的环境对象，这个对象可能是 env 本身，也可能是已经被封装过的 env 类。在执行环境的功能时，Wrapper 有两个选择，如果实现了以下画线开始的同名方法，就会调用这个函数，否则就会调用 env 中对应的函数。以前面提到的“step”功能为例，如果 Wrapper 内实现了名为“_step”的函数，那么在调用 Wrapper 的 step 函数时就会执行“_step”方法，如果没有则执行 env 的 step 方法。

我们来看一个官方给出的例子：**ObservationWrapper**。这个类可以对观测值做额外的处理，例如为任务增加难度、去掉部分的观测值，或者为观测值增加随机噪声等，这个类的核心方法如下所示：

```
def _step(self, action):
    observation, reward, done, info = self.env.step(action)
    return self.observation(observation), reward, done, info

def observation(self, observation):
    return self._observation(observation)

def _observation(self, observation):
    # 我们实现的代码
```

通过实现_observation 方法中的内容，我们就可以操纵游戏的观测值，从而对游戏增加难度。后面还会介绍一些与 Wrapper 相关的实现。

5.1.3 利用 Gym 框架实现一个经典的棋类游戏：蛇棋

下面以一个实际问题为例，剖析如何用 Gym 框架实现一个全新的任务。笔者是一名“85 后”，年幼时曾接触过一种被称为蛇棋的棋类游戏。这个游戏是一个棋盘游戏，棋盘上有许多小格子，每个格子代表一个位置，除此之外，游戏还需要几个棋子和一个骰子。棋盘的样子如图 5-5 所示。

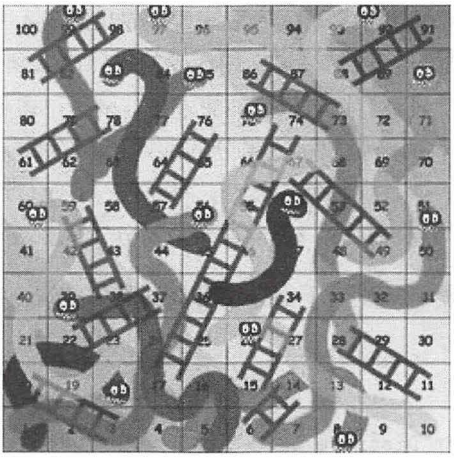


图 5-5 蛇棋的棋盘

蛇棋的玩法如下。

（1）玩家每人拥有一个棋子，出发点在图中标为“1”的格子处。

（2）依次掷骰子，根据骰子的点数将自己的棋子向前行进相应的步数。假设笔者的棋子在“1”处，并且投掷出“4”，则笔者的棋子就可以到达“5”的位置。

（3）棋盘上有一些**梯子**，它的两边与棋盘上的两个格子相连。如果棋子落在其中一个格子上，就会自动走到梯子对应的另一个格子中。以图 5-5 所示的棋盘为例，如果笔者的棋子在“1”处，并且投掷出“2”，那么棋子将到达“3”处，由于此处有梯子，棋子将直接前进到梯子的另一段——“20”的位置。

（4）最终的目标是到达“100”处，如果在到达时投掷的数字加上当前的位置超过了 100，那么棋子将首先到达 100，剩余的步数将反向前进。例如，笔者的棋子在“99”处，如果投掷出“3”，则先前进 1 步到达 100，再后退 2 步，到达“98”处。“98”处恰好还有一个梯子，于是棋子将顺着梯子到达“81”处。

这个游戏是强化学习的一个小案例。蛇棋中的棋盘和游戏规则是既定的，胜利规则也是既定的，游戏中唯一存在变化的就是掷骰子。虽然我们通常认为掷骰子是一个很简单而且随机性很大的操作，但实际上，每个人掷骰子时的状态都不相同，有些人掷骰子时要气沉丹田，有些要大喝一声，有些小心翼翼……因此我们可以认为掷骰子的方式不同，骰子各个面出现的概率就不同。

基于这样的想法，我们认为玩家可以通过使用不同掷骰子的方法来控制游戏。假设玩家的手法只包含两种：一种可以均匀投掷出 1~6 这 6 个数字，另一种可以均匀投掷出 1~3 这 3 个数字。这样玩家相当于在每一个状态下有两个选择。

接下来看看游戏的最终获胜条件。一般来说，游戏是多人一起玩的，那么谁先到达“100”处，谁就获得了胜利。换句话说，用最少的投掷次数到达“100”处是每一个玩家的目标。为了达到这个目标，玩家最好能更多地“乘坐”梯子上升，以便快速到达终点。一般来说，游戏喜欢用一些数字或得分的形式记录给予玩家回报，这里做一个约定，玩家投掷一次骰子，如果没有到达终点，将获得 −1 分；到达终点后，将一次性得到 100 分。这样先到达终点的玩家一定会得到最高分，也就相当于这个人获得了胜利。

相信读者已经清楚游戏规则，下面介绍游戏的具体实现。我们给出下面这段代码：

```
import numpy as np
import gym
from gym.spaces import Discrete

class SnakeEnv(gym.Env):
    SIZE=100

    def __init__(self, ladder_num, dices):
        self.ladder_num = ladder_num
        self.dices = dices
        self.ladders = dict(np.random.randint(1, self.SIZE, size=(self.
        ladder_num, 2)))
        self.observation_space=Discrete(self.SIZE+1)
        self.action_space=Discrete(len(dices))

        for k,v in self.ladders.items():
            self.ladders[v] = k
            print 'ladders info:'
            print self.ladders
            print 'dice ranges:'
            print self.dices
        self.pos = 1

    def reset(self):
        self.pos = 1
        return self.pos

    def step(self, a):
        step = np.random.randint(1, self.dices[a] + 1)
```

```
        self.pos += step
        if self.pos == 100:
            return 100, 100, 1, {}
        elif self.pos > 100:
            self.pos = 200 - self.pos

        if self.pos in self.ladders:
            self.pos = self.ladders[self.pos]
        return self.pos, -1, 0, {}

    def reward(self, s):
        if s == 100:
            return 100
        else:
            return -1

    def render(self):
        pass
```

在这段代码中，构造函数需要传入两个参数：梯子数量和不同投掷骰子方法的最大值。我们用一个 dict 存储梯子相连的两个格子的关系，用一个 list 保存可能的骰子的可投掷最大值；reset 方法将 pos 设置为 1，也就是游戏开始的位置；step 完成一次投掷，参数 a 表示玩家将采用何种手法。完成位置的更新后，函数将返回玩家的新位置、得分和其他信息。

我们可以利用这个环境进行游戏，设定游戏中共有 10 个梯子，两种投掷骰子的方法分别可以投掷 $[1,3]$ 和 $[1,6]$ 的整数值，玩家将完全使用第一种策略进行游戏，那么代码如下：

```
if __name__ == '__main__':
    env = SnakeEnv(10, [3,6])
    env.reset()
    while True:
        state, reward, terminate, _ = env.step(0)
        print reward, state
        if terminate == 1:
            break
```

对应的输出为：

```
ladders info:
{2: 99, 67: 73, 54: 82, 73: 67, 10: 58, 43: 86, 45: 95, 93: 23, 50: 59,
17: 92, 82: 54, 99: 2, 86: 43, 23: 93, 26: 73, 59: 50, 92: 17, 58: 10, 95:
 45}
dice ranges:
[3, 6]
-1 7
-1 9
……(以下省略若干中间结果)
-1 90
-1 96
-1 97
100 -1
```

最终玩家完成了一次游戏，在这次游戏中，玩家共进行了 294 次投掷才到达终点，最终得分 −193 分。从上面的梯子情况可以看出，有些梯子真的给玩家制造了不小的麻烦：

```
99: 2
95: 45
92: 17
```

这些梯子既会让玩家迅速靠近终点，又会让玩家迅速远离终点，一旦玩家多次掉入这种陷阱，游戏时间就会变长。好胜心强的读者肯定在想，刚才的游戏中玩家只使用了一种手法，要是两种手法并用，甚至更多的手法并用，一定能更快地完成游戏。事实是这样吗？第 6 章和第 7 章我们将使用强化学习中的基本算法在这个例子上继续尝试。

5.2 Baselines

Baselines 是 OpenAI 主导的一个开源项目，主要的编程语言为 Python 3，它实现了当前强化学习界一些经典的算法，并将这些算法和自家的 Gym 环境结合，形成一个非常好的生态环境。如果说 Gym 实现了环境相关的功能，那么 Baselines 就实现了 Agent 相关的功能。这使得研究人员和学生能够在非常短的时间内体验经典强化学习算法的效果，并了解这些算法的内容。项目的代码结构如图 5-6 所示，其中包含了子项目对应的算法名称和本书对算法介绍的章节。

由于 Baselines 中的算法将在本书后面的章节中逐一介绍，因此这里就先略去不谈。本节重点介绍这个项目中的一些辅助功能，其中包括它为 Gym 和 TensorFlow 开发的一

些辅助功能，也就是通用类代码 common 中的一部分内容。实际上，前面的章节已经介绍了其中部分内容，在 3.3 节中，我们介绍了共轭梯度法在代码中的实现，对应的代码在 common/cg.py 中；在 4.3 节中，我们介绍了基于 MPI 协议的优化器，对应的代码在 common/mpi_adam.py 中。本节将介绍其他比较值得关注的部分。

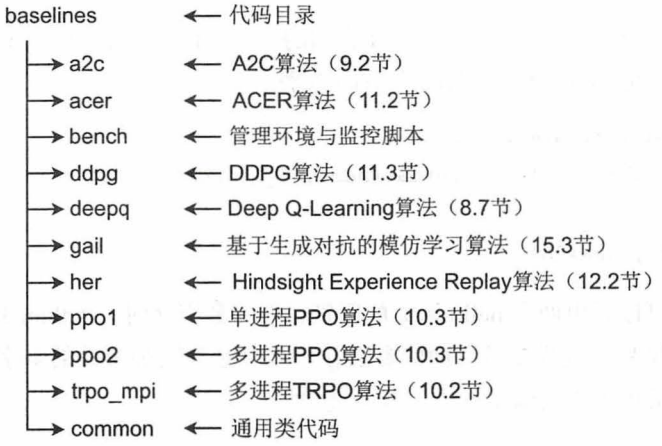


图 5-6 Baselines 的代码结构

5.2.1 Baselines 中的 Python 3 新特性

这个项目比较好地完成了针对 Gym 环境平台的功能扩展和经典算法的实现。当然，除了为用户带来便利，它对开发环境也做了一定的限制。项目基于 Python 3 开发完成，同时深度集成了 TensorFlow 的一些特性，这需要读者对这两部分知识有一定的了解。TensorFlow 的内容在前面已经介绍过，下面笔者简单介绍这个项目中用到的两个 Python 3 的特性，使用 Python 2 的读者可以通过下面的内容快速填补对新特性了解的空白。

第一个特性就是在函数参数中加入星号（*），在 Python 3 中，星号有了许多新的作用，它的出现使我们的代码更简洁易读。我们常常会定义带有很多参数的函数，代码如下：

```
def hello2(name, age, sex):
    print ('hello {},{},{}'.format(name, age, sex))
```

使用这样的函数需要每次填入三个参数，有时是比较麻烦的，于是我们会给其中一些参数赋予默认值，然后代码就变成了这样：

```
def hello2(name, age=20, sex=1):
```

```
    print ('hello {},{},{}'.format(name, age, sex))
```

虽然不用填写复杂的参数，但是某一天在阅读代码时，读者可能会突然发现这样的代码：

```
hello2('world', 25)
```

时间一长，就会记不清楚 25 是干什么用的。为了解决这个问题，Python 3 中要求某些参数必须附加参数名称，增强代码的可读性：

```
def hello3(name, *, age=20, sex=1):
    print ('hello {},{},{}'.format(name, age, sex))

hello3('world', age=25)
```

从代码中可以看出两个 hello 函数的区别，在参数定义时，Python 3 可以在指定的位置加入一个星号，这样星号后面的参数在传入时必须附加参数的名字，否则就会报错，如果我们采用下面的调用方式，就会报错：

```
IN:  hello3('world', 25)
OUT: TypeError: hello3() takes 1 positional argument but 2 were given
```

第二个特性同样和星号有关。有时我们会遇到返回值比较多的函数，代码如下所示：

```
def func():
  # 省略一些构造内容
  return x, feat1, feat2, feat3, feat4, y
```

我们凭直觉看出返回值分为 3 类：x、featX 和 y。如果使用 Python 2，则需要先用一个变量接收返回值，然后将返回值拆开：

```
data = func()
x = data[0]
y = data[-1]
feat = data[1:-1]
```

而在 Python 3 中，我们只需要写成：

```
x, *feat, y = func()
```

除第一个和最后一个返回值外，中间的所有返回值都会保存到 feat 变量中，代码和之前相比显得更整洁。

除这两个语法外，Baselines 中还使用了其他新语法，但它们都比较直观，容易理解，因此本节不再赘述。

5.2.2 tf_util

Baselines 项目对 TensorFlow 的一些基本功能做了封装，为了在后面的章节详细介绍其中提供的代码，在此对项目中常见的 TensorFlow 封装内容做一个介绍。这部分的代码在 common/tf_util.py 中。

tf_util 也对 Session 管理部分进行了实现。由于强化学习的算法中可能会同时出现多个模型，而且模型的优化和预测过程相互依赖，因此 Session 无法被某一个模型独占。代码采用全局单例 Session 的设计方法，将 Session 对象设计成一个全局对象，统一通过一个方法调用获取：

```
def get_session():
    """Returns recently made Tensorflow session"""
    return tf.get_default_session()

def make_session(num_cpu):
    """Returns a session that will use <num_cpu> CPU's only"""
    tf_config = tf.ConfigProto(
        inter_op_parallelism_threads=num_cpu,
        intra_op_parallelism_threads=num_cpu)
    return tf.Session(config=tf_config)
```

除此之外，它还实现了初始化、取值等常见的操作，让这个普通的 Session 操作更简便。

代码中另一个比较有特点的部分是 Theano-like 函数。TensorFlow 的计算图在操作时粒度比较小，在面对比较复杂的模型时，太多细粒度的 Op 会让代码变得很乱。例如，读者在后续章节中会接触到，一个 Off-Policy 的 Actor Critic 的算法可能包含 Actor、Critic、Target Actor 和 Target Critic 四个模型，这些模型都包含预测和更新的操作，这些操作交织在一起就会显得比较杂乱。为此，tf_util 中实现了这样一个功能，不论模型的定义多么复杂，我们可以将一个模型的输入、输出和更新集合成为一个整体，这个整体被称为函数，未来的交互只要通过这个函数的接口即可。

函数的主要属性如下。

（1）inputs：函数的输入 Tensor。

（2）outputs：函数的输出 Tensor。

（3）updates：函数的更新 Op。

（4）givens：函数输入的默认值。

所以函数的主要执行过程就是：

```
def __call__(self, *args, **kwargs):
    feed_dict = {}
    for inpt, value in zip(self.inputs, args):
        self._feed_input(feed_dict, inpt, value)
    for inpt in self.givens:
        feed_dict[inpt] = feed_dict.get(inpt, self.givens[inpt])
    results = get_session().run(self.outputs_update, feed_dict=feed_dict)
        [:-1]
    return results
```

其中还有一些错误检查的代码，在此略过。我们可以使用它的 Function 功能完成一个小的示例：可以将 4.1 节中示例的模型修改成一个函数。在 4.1 节中实现了基于神经网络完成异或判断的模型，我们可以定义输入为模型的输入，输出为模型的输出和 loss 值，更新为一轮训练后的参数更新，于是代码可以写作：

```
import tensorflow as tf
x = tf.placeholder(tf.float32, shape=[None, 2], name='input')
y = tf.placeholder(tf.float32, shape=[None, 1], name='output')
# 省略模型构建过程
loss = tf.nn.l2_loss(z2 - y)
opt = tf.train.GradientDescentOptimizer(0.05)
train_op = opt.minimizer(loss)
model = function([x,y],[z2, loss],updates=[train_op])
for x_data, y_data in data:
    model(x_data, y_data)
```

只要像调用函数一样调用这个类，就可以执行输出和更新两个工作。以上就是 tf_util 中的一些方便开发的函数，读者也可以像这样根据自己的需要开发一个常用的工具库，提高自己的工作效率。

5.2.3 对 Gym 平台的扩展

Gym 平台在集成了众多环境后，并未对每一个环境提供更多深度定制的功能，玩家需要根据自己的情况进行开发。例如对 Atari 游戏来说，很多论文的实验章节都明确写道，在实验时，为了确保一定的随机性，游戏开始的前 30 帧将采取随机操作。由于 Gym 中没有提供这样的特性，开发者只能靠自己完成。但是 Baselines 开发了很多与

书籍是人类进步的阶梯

Atari 游戏有关的环境封装，这些功能在文件 common/atari_wrappers.py 中。创建 Atari 游戏环境时，可以调用文件中的这个函数：

```
def make_atari(env_id):
    env = gym.make(env_id)
    assert 'NoFrameskip' in env.spec.id
    env = NoopResetEnv(env, noop_max=30)
    env = MaxAndSkipEnv(env, skip=4)
    return env
```

从代码中可以看出，除调用 Gym 的接口创建游戏环境外，它还运用了 Gym 中的 Wrapper 功能，为当前的环境类增添新的功能，使得它在原本的功能上可配置地增添新的功能。

封装的第一个类 **NoopResetEnv** 就实现了上面提到的“前 30 帧采取随机操作”的功能。它的核心代码（经过修改）如下所示：

```
def _reset(self, **kwargs):
    # self.env是构造函数传入的环境对象
    self.env.reset(**kwargs)
    noops = 30
    obs = None
    for _ in range(noops):
        obs, _, done, _ = self.env.step(self.noop_action)
        if done:
            obs = self.env.reset(**kwargs)
            return obs
```

从代码中可以看出，在用户对它执行 reset 操作时，它首先让自己类内的 env 对象执行 reset 方法，然后执行 30 轮 noop 操作，完成这些之后才将观测到的状态返回。有了这样的工具类，我们就不需要为这些琐碎的内容分心了。

除了这一层封装，环境还经过 **MaxAndSkipEnv** 的封装。这个类又实现了怎样的功能呢？它的核心代码如下所示：

```
def _step(self, action):
    total_reward = 0.0
    done = None
    for i in range(self._skip):
        obs, reward, done, info = self.env.step(action)
        if i == self._skip - 2: self._obs_buffer[0] = obs
```


书籍是人类进步的阶梯

```
            if i == self._skip - 1: self._obs_buffer[1] = obs
                total_reward += reward
                if done:
                    break
        max_frame = self._obs_buffer.max(axis=0)
        return max_frame, total_reward, done, info
```

从代码中可以看出，每一次 Agent 产生一个行动，并将行动传递给环境，而环境并非只对内部的环境类调用一次 step 方法，而是调用 self._skip 次，每一次都使用相同的行动进行操作。除此之外，它还将保存最后两个时刻的观测值，并取出两个时刻的最大值进行返回。Atari 游戏每秒的刷新帧数为 60 次，我们固然可以让 Agent 在每一帧完成不同的行动，但这样的操作频率远超人类，不但给算法的实时性带来负担，而且这样高频率的操作也没有必要。因此，将机器的操作频率尽可能地靠近人类显得更有意义。

有了这些功能，我们可以将所有的精力放在 Agent 的训练和执行上，节省了不少关注环境的精力。

5.3 总结

本章介绍了与强化学习有关的开发环境，总结如下。

（1）Gym 集成了众多强化学习环境，使用户可以方便地使用各种环境训练模型。

（2）Baselines 实现了一些方便开发的辅助函数和经典的强化学习算法，用户可以直接尝试这些算法的效果。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

6 强化学习基本算法

从本章开始，我们正式介绍强化学习，接下来的章节将充满公式与定义，相信对读者来说是一个挑战。在第 1 章，我们已经介绍了强化学习的基本概念，在 5.1 节介绍了基于 Gym 的强化学习的基本流程。本章进一步形式化这个过程，站在数学的角度分析这个问题的解决方案。对强化学习来说，一般以马尔可夫决策过程（Markov Decision Process，MDP）这样一个模型作为形式化的手段。本章将介绍 MDP 的原理，同时介绍基于 MDP 的策略学习算法：策略迭代、价值迭代和泛化迭代。

6.1 马尔可夫决策过程

MDP 是当前强化学习理论推导的基石，通过这套框架，强化学习的交互流程可以很好地以概率论的形式表示出来，解决强化学习问题的关键定理也可以依此表示出来。从此我们就可以把目光集中在这些抽象的数学符号上，并使用数学工具推演性质帮助强化学习的性质。本节的重点就是介绍 MDP 和相关算法。

6.1.1 MDP：策略与环境模型

5.1 节中提到了蛇棋这个游戏，我们先来关注蛇棋的一个关键问题：哪些因素决定了蛇棋最终获得分数的多少？其实只有两个因素：选择什么样的手法投掷和投掷出的数目。第一个因素是玩家可以决定的，也是玩家唯一可以决定的；第二个因素是玩家无法确定的，它只受环境的随机性控制。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

如果再进一步看，有哪些结果/状态决定了最终的得分呢？那就是每一步走到位置和每一步选择的投掷手法了。还以蛇棋为例，我们用 s_t 表示 t 时刻游戏状态的观测值，也就是行走到的位置，用 a_t 表示 t 时刻选择的手法，那么游戏过程就可以用一条状态–行动链条表示：

$$\{s_0, a_0, s_1, a_1, \ldots, s_{t-1}, a_{t-1}, s_t\}$$

这个链条包含了两种状态转换，一种是从状态到行动的转换，另一种是从行动到状态的转换，这两种转换分别对应了前面提到的两个决定因素。第一种转换是由 Agent 的**策略**决定的，第二种转换是由**环境**决定的。下面就来看看这两种转换的具体形式。

作为第一种转换——策略，指的是 Agent 根据当前的状态选择“自认为”最好的行动方式。如果用严谨的方式表述，策略是一种映射，它将环境的状态值 s_t 映射到一个行动集合的概率分布或概率密度函数上。对于蛇棋这个问题，我们可以认为 Agent 会在内心对每一个行动进行衡量，并最终选择评价最高的行动；如果我们认为对于每一个行动，Agent 都有一定的概率去执行，且行动的评价越高，行动产生的概率越大，那么玩家就会选择概率最高的一种行动，形式化来说就是在时间 t 的时刻，选择公式

$$a_t^* = \text{argmax}_{a_{t,i}} p(a_{t,i} | \{s_0, a_0, \ldots, s_t\})$$

的结果作为行动。可以看到，这个条件概率还是比较复杂的，行动依赖的信息比较多，需要对其做一定的化简才能方便计算。对蛇棋来说，可以使用蛇棋中状态之间无依赖的性质进行化简，即当前时刻选择什么行动只和当前的状态有关，和前面的状态与行动无关。这一点比较容易理解，如果 Agent 在“55”这个位置，那么它只需要考虑处于位置“55”之后的行动，至于如何从“1”走到“55”，已经不重要了，Agent 的目标十分明确，就是尽快从“55”移动到“100”，于是上面的公式就变为

$$\text{argmax}_{a_i} p(a_i | S_t)$$

这样问题就得到了简化，实际上这步化简用到了序列的马尔可夫性，也就是说下一时刻的行动只和当前时刻的状态有关，和当前时刻之前的状态无关。以蛇棋为例，这个公式还可以进一步化简，由于行动的集合是离散有限的，可以把选择行动的问题变成一个多分类问题，这样就把策略选择问题变成了一个熟悉的问题。

第二种转换是环境的状态转换。当 Agent 完成行动后，环境会受到影响并完成状态的转换。这里也非常明显，下一步的状态只受前一步状态影响，不受更前面的状态影响，于是这里的状态转换就能以概率的形式表现为

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$p(\boldsymbol{S}_{t+1}|\boldsymbol{S}_t, \boldsymbol{a}_t)$$

假设 Agent 在“55”这个位置，如果它选定使用投掷“1~3”数字的手法，假设骰子投掷结果是“均匀”的，那么接下来它将等概率地出现在“56”、“57”和“58”这三个格子中。但是由于梯子的存在，落在这三个格子中的棋子可能会到达其他位置，例如通过梯子，最终到达了“56”、“40”和“80”处，那么这三个格子的概率分别为三分之一。所以实际中需要考虑棋盘中每一个格子的概率，即使绝大多数格子的概率为 0。这部分的信息实际上属于环境内部的信息，在蛇棋这个问题中是公开的，而在一些问题中却是不可见的。在本章我们认为状态转换都是已知的，以便我们做接下来的推导。

了解了这两个过程，就可以重新研究 MDP 这个词了。马尔可夫决策过程包含以下三层含义。

（1）“马尔可夫”表示了状态间的依赖性。当前状态的取值只和前一个状态产生依赖，不和更早的状态产生联系。虽然这个条件在有些问题上有些理想，但是由于它极大地简化了问题，所以人们通常会选择使用它。

（2）“决策”表示了其中的策略部分将由 Agent 决定。Agent 可以通过自己的行动改变状态序列，和环境中存在的随机性共同决定未来的状态。

（3）“过程”表示了时间的属性。如果把 Agent 和环境的交互按时间维度展开，那么 Agent 行动后，环境的状态将发生改变，同时时间向前推进，新的状态产生，Agent 将获得观测值，于是新的行动产生，然后状态再更新……

6.1.2 值函数与 Bellman 公式

前面介绍了 MDP 的基本形式，我们发现游戏的关键在于策略，也就是如何做出决策与执行行动。在理想状态下，每一个行动都要为最终的目标——最大化长期回报努力，那么理论上只要能够找到一种方法，量化每一个行动对实现最终目标贡献的价值，Agent 就可以根据这些量化指标做出明智的判断。所以接下来一个重要的工作就是量化这些价值。

幸运的是，环境已经帮我们完成了最重要的一步——它提供了可以量化的某一时刻的回报值 $\boldsymbol{r}$。虽然这只是局部的回报，和我们最终的目标有些不同，但我们可以利用它，将其扩展，使之成为我们的目标。前面已经提到关于蛇棋的回报规则，当我们落在非终点的位置上时，$\boldsymbol{r}=-1$；当我们落在终点时，$\boldsymbol{r}=100$。前面提到我们的策略等同于最大化下面这个公式

$$\text{maximize} \sum_t \boldsymbol{r}_t$$

書籍是人类进步的阶梯

仅供非商业用途或交流学习使用

这个公式并不容易计算，它的困难反映在两方面，其中一个在计算的时间上。如果游戏的时间维度是有限的，也就是说，我们可以在有限步内完成游戏，那么这个公式虽然复杂，但至少是可以计算的；若游戏可以无限进行下去呢？例如在“94”～“99”这6个位置全架设向小数字位置移动的梯子，这样 Agent 将永远无法正好到达“100”这个位置，虽然问题变得不再有意义，但我们也不能直接用上面的公式了。

为了解决这个问题，使这个无穷数列的和收敛，我们要降低未来回报对当期的影响，也就是对未来的回报乘以一个打折率，使长期回报变得更有意义。所以修正后的公式变为

$$\sum_{t=0} \gamma^t \boldsymbol{r}_t$$

为什么要做这样的修改呢？这个公式的灵感实际上来源于生活，最直接的对应就是商业银行的储蓄了。我们知道把钱存入银行，银行是可以支付利息的（极端情况下还会出现负利息）。假设银行的年利率是 10%（现实中恐怕很难见到），于是我们存入100 元，一年后它就变成了 110 元。如果我们希望一年后它变成 100 元，那么现在需要存入多少钱呢？运用初等数学，令存入的钱数为 X，可以得到

$$X \times (1 + 0.1) = 100$$

得到的结果大概是 90.9 元，这个数字比 100 元小。也就是说，考虑到货币的时间效果，今天的 90.9 元和 1 年后的 100 元是等价的，如果将这 90.9 元存入银行两年，会使它变得更值钱。利用这个思想，既然现在的资产会随着时间不断提升价值，那么把未来的回报换算到当下的确是要打折的。这个打折率一般来说小于 1，这样一来长期回报的这个数列的和就变得有界了，也就可以计算出具体的值了。

于是我们正式定义另外一个变量——**长期回报**。所谓长期回报，是将当前状态之后所有的回报取出，分别乘以对应的打折率，然后加起来得到一个汇总的值

$$\mathrm{Ret}_t = \sum_{k=0} \gamma^k \boldsymbol{r}_{t+k+1}$$

解决了长期回报的可表示问题，另一个困难也出现了，用求和的形式表示长期回报显得有些复杂，而且在行动执行时我们并不知道未来的行动情况，例如投掷骰子的具体数值，那么能不能用其他方法表示对这个公式的估计呢？这时我们就需要另一个定义：策略的**价值**。前面提到了基于 MDP 形成的交互序列，其中从状态到行动的转换可以通过策略确定，而由于环境的原因，从行动到下一个状态的转换有时并不能确定。因此在衡量价值时，我们需要考虑每一种状态转换的影响，这就需要基于状态转换求

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

解长期回报的期望。令 τ 为根据策略和状态转换采样得到的序列，那么价值的公式定义就可以写成

$$v_\pi(s_t) = E_{s,a\sim\tau}[\sum_{k=0} \gamma^k r_{t+k+1}]$$

根据 MDP 的模型形式，价值函数（一般称为“值函数”）可以分为两种类型。

（1）状态值函数 $v_\pi(s)$：也就是已知当前状态 s，按照某种策略行动产生的长期回报期望。

（2）状态–行动值函数 $q_\pi(s,a)$：也就是已知当前状态 s 和行动 a，按照某种策略行动产生的长期回报期望。

实际上，即使采用这样的表达方式，计算价值仍然是个很困难的事情，如果要计算从某个状态出发的值函数，相当于依从某个策略把所有从这个状态出发的可能路径走一遍，将这些路径的长期回报依概率求期望：

$$\begin{aligned} v_\pi(s_t) &= E_\tau[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}] \\ &= \sum_\tau p(\tau) \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned}$$

其中 τ 表示从状态 s_t 出发的某条路径。由于我们将强化学习的过程表示成马尔可夫决策过程，于是路径部分可以展开为

$$v_\pi(s_t) = \sum_{(s_t,a_t,\ldots)\sim\tau} \pi(a_t|s_t)p(s_{t+1}|s_t,a_t)\ldots\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

这样的公式形式显然不够优雅，于是我们使用高中时使用的代换消元法，就可以得到

$$\begin{aligned} v_\pi(s_t) &= \sum_\tau \pi(a_t|s_t)p(s_{t+1}|s_t,a_t)\ldots\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= \sum_{a_t} \pi(a_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t) \sum_{(s_{t+1},a_{t+1},\ldots)\sim\tau'} \pi(a_{t+1}|s_{t+1})\ldots\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= \sum_{a_t} \pi(a_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t) \sum_{(s_{t+1},a_{t+1},\ldots)\sim\tau'} \pi(a_{t+1}|s_{t+1})\ldots[r_{t+1}+\sum_{k=1}^{\infty} \gamma^k r_{t+k+1}] \\ &= \sum_{a_t} \pi(a_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t,a_t)[r_{t+1}+ \sum_{(s_{t+1},a_{t+1},\ldots)\sim\tau'} \pi(a_{t+1}|s_{t+1})\ldots\sum_{k=1}^{\infty} \gamma^k r_{t+k+1}] \end{aligned}$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

将下一时刻的状态价值展开，可以得到

$$v_\pi(\boldsymbol{s}_{t+1}) = \sum_{(\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1},\ldots)\sim\tau'} \pi(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1})p(\boldsymbol{s}_{t+2}|\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1})\cdots\sum_{k=1}^{\infty}\gamma^k \boldsymbol{r}_{t+k+1}$$

两式融合，再经过一些变换，可以得到

$$v_\pi(\boldsymbol{s}_t) = \sum_{\boldsymbol{a}_t}\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)\sum_{\boldsymbol{s}_{t+1}}p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t,\boldsymbol{a}_t)[\boldsymbol{r}_{t+1}+v_\pi(\boldsymbol{s}_{t+1})]$$

通过这样的计算，我们发现状态值函数可以以递归的形式表示。假设值函数已经稳定，任意一个状态的价值可以由其他状态的价值得到，这个公式就被称为贝尔曼公式（Bellman Equation，以下简称 Bellman 公式），是后面章节进行策略求解的基础公式之一。之所以称为“之一”，是因为状态-行动值函数同样有一个类似的公式

$$q_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t) = \sum_{\boldsymbol{s}_{t+1}}p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t,\boldsymbol{a}_t)\sum_{\boldsymbol{a}_{t+1}}p(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1})[\boldsymbol{r}_{t+1}+q_\pi(\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1})]$$

这个公式的证明方式与上面的公式类似，这里不再赘述。这两个公式可以说是整个强化学习理论的基石，请读者务必对其深入理解。接下来，我们以图的形式对两个公式进行阐述。

图 6-1 以树的形式展示了 MDP 的扩展过程。问题的开始节点为 $\boldsymbol{s}_0$，当策略 π 固定时，对应的行动将根据策略以一定的概率产生，对应的状态转移也会以一定的概率产生。

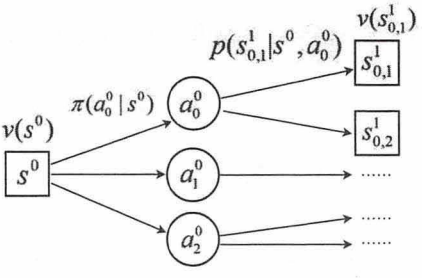


图 6-1　MDP 的扩展形式与状态价值

图 6-1 中的变量名称比较复杂，其中 $\boldsymbol{s}^0$ 表示 $t=0$ 时刻的状态，$\boldsymbol{a}_0^0$ 表示 $t=0$ 时刻的 0 号行动，而 $\boldsymbol{s}_{0,2}^1$ 表示 $t=1$ 时刻从上一时刻 $\boldsymbol{a}_0^0$ 得到的第 2 个状态。在状态转移过程中，我们有可能在不同时刻遇到同一个状态，那么这两个状态的价值相等吗？

仅供非商业用途或交流学习使用

为了证明这一点，我们要给出一个上面证明中没有明确提到的假设：**时间无关性**。时间无关性是指 MDP 经过不断演化，最终会使不同时刻下相同状态的价值相等。举例来说，$t=1$ 时刻的 $v(s)$ 和 $t=3$ 时刻的 $v(s)$ 相等。为了达到这个假设，我们认为 MDP 的状态转移过程持续时间足够长，最终每个状态、行动的转移进入了稳定的状态，从图 6-1 中可以看出，从下一时刻的状态 s_1 开始的价值可以用 $v(s_1)$ 表示，于是根据它们的转移概率，同样可以得到 Bellman 公式的形式。图 6-2 则是状态–行动值函数的表示形式，两者在假设和推演上是类似的。

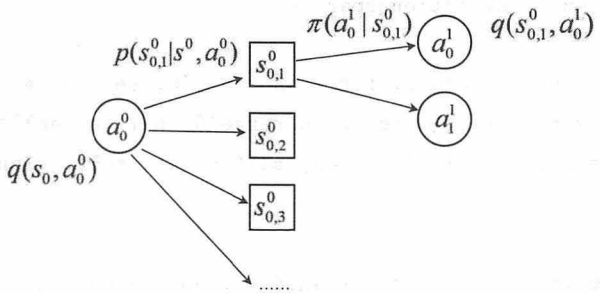


图 6-2 MDP 的扩展形式与状态--行动价值

看到这里，读者一定会想，既然 Bellman 公式具有递归定义的特点，那么能不能利用这个特点求解出值函数呢？问题开始时，我们并不知道值函数，而求解值函数需要知道其他时刻的值函数，这似乎为我们求解价值制造了障碍，那么该如何解决问题呢？从本章的 6.2 节开始我们会给出答案。

6.1.3 “表格式” Agent

前面介绍了太多公式推导，我们并没有一个对 MDP 具体实现的概念，那么现在就来看看这些内容对应的代码实现，本节要实现“表格式” Agent 需要的基本数据结构。什么是“表格式”？以蛇棋问题为例，棋盘上有 100 个位置，所以问题一共有 100 个离散的状态；投掷骰子的手法也是有限的，因此也可以通过离散的方式表示出来。所以对上面提到的所有实体（状态、行动、策略、状态转移概率），都可以以 N 维张量的形式表示。例如，策略 $\pi(\boldsymbol{a}|\boldsymbol{s})$ 是一个条件概率分布，如果以“表格”的形式表示，则这个条件概率可以由一个 $|\boldsymbol{S}| \times |\boldsymbol{A}|$ 的矩阵表示，每一个数值都处于 0 到 1 的范围内，每一行的数值相加之和为 1。而对于状态转移 $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ 来说，如果以“表格”的形式表示，那么这个条件概率分布就可以由一个 $|\boldsymbol{S}| \times |\boldsymbol{S}| \times |\boldsymbol{A}|$ 的张量表示。

除了“表格式”这个约束，还需要另外一个前面提到的约定：环境的状态转移概率需要对 Agent 公开，这样 Agent 就能利用这些信息做出更好的决策。对蛇棋来说，如果

仅供非商业用途或交流学习使用

知道骰子的每一面朝上的概率是均匀的，以及棋盘上的每一个梯子都是可见的，就可以计算出状态转移概率。

所以我们现在实现一段简单的代码，它会根据环境的信息生成问题中所有实体的“表格式”数据结构，Agent 代码的基本结构如下所示：

```
class TableAgent(object):
    def __init__(self, env):
        self.s_len = env.observation_space.n
        self.a_len = env.action_space.n

        self.r = [env.reward(s) for s in range(0, self.s_len)]
        self.pi = np.array([0 for s in range(0, self.s_len)])
        self.p = np.zeros([self.a_len, self.s_len, self.s_len], dtype=np.
        float)

        ladder_move = np.vectorize(lambda x: env.ladders[x] if x in env.
        ladders else x)

        for i, dice in enumerate(env.dices):
            prob = 1.0 / dice
            for src in range(1, 100):
                step = np.arange(dice)
                step += src
                step = np.piecewise(step, [step > 100, step <= 100],
                    [lambda x: 200 - x, lambda x: x])
                step = ladder_move(step)
                for dst in step:
                    self.p[i, src, dst] += prob
        self.p[:, 100, 100]=1
        self.value_pi = np.zeros((self.s_len))
        self.value_q = np.zeros((self.s_len, self.a_len))
        self.gamma = 0.8

    def play(self, state):
        return self.pi[state]
```

可以看出，这个 Agent 中保存了以下信息。

- 蛇棋的状态、行动数目：$|S|$、$|A|$。

仅供非商业用途或交流学习使用

- 蛇棋中的回报记录 $\boldsymbol{r}$：一个一维数组。
- 蛇棋的策略 $\boldsymbol{\pi}$：一个二维数组（$|\boldsymbol{S}|, |\boldsymbol{A}|$），存储 $\pi(\boldsymbol{a}|\boldsymbol{s})$ 的结构。
- 蛇棋中的状态转移情况 p：这是一个三维的数组（$|\boldsymbol{A}|, |\boldsymbol{S}|, |\boldsymbol{S}|$），存储 $p(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ 的结构。
- 状态价值函数 $v(\boldsymbol{s})$。
- 状态-行动值函数 $q(\boldsymbol{s}, \boldsymbol{a})$。
- 打折率 γ。

到此为止，初始化结束。这些结构将在 6.2 节派上用场。

6.2 策略迭代

6.1 节我们使用 MDP 的框架定义了一系列的概念，其中包括十分重要的两种值函数。如果使用值函数重新定义强化学习的目标，我们可以得到：强化学习就是找到最优的策略，使每一个状态的价值最大化，这相当于求解

$$\boldsymbol{\pi}^* = \text{argmax}_{\boldsymbol{\pi}} v_{\boldsymbol{\pi}}(\boldsymbol{s}), \forall \boldsymbol{s}$$

而对于每一个状态对应的行动，我们希望找到使其价值最大化的行动

$$\boldsymbol{a}^* = \text{argmax}_{\boldsymbol{a}} q_{\boldsymbol{\pi}^*}(\boldsymbol{s}, \boldsymbol{a})$$

可以看出为了求出最终的结果，我们需要同时更新交织在一起的策略与价值，本节将详细介绍解决这个问题的算法之一：**策略迭代法（Policy Iteration）**。

6.2.1 策略迭代法

在本节开篇提到的计算思路中，我们发现如果想知道最优的策略，就需要能够准确估计值函数。然而想准确估计值函数，又需要知道最优策略，数字才能够估计准确。所以实际上这是一个“鸡生蛋还是蛋生鸡”的问题。碰上这样无解的问题，往往需要采取一些“曲线救国”的方法。那么，能不能把这个问题考虑成一个迭代优化的问题，通过一轮一轮的计算逐渐接近最优的结果呢？答案是可以的。

于是我们提出如下计算思路。

（1）以某种策略 π 开始，计算当前策略下的值函数 $V_\pi(s)$。

（2）利用这个值函数，找到更好的策略 π^*。

（3）用这个策略继续前行，更新值函数，得到 $V'_\pi(s)$，然后不断迭代……

这样经过若干轮计算，如果一切顺利，我们的策略会收敛到最优的策略，如果能够判断出策略已经收敛，问题也就得到了解答。当然，目前我们还没有理论证明，因此这个思路也可能是错的，但我们还是先试着实践这个思路。

为了验证结果的正确性，我们需要将蛇棋的难度降低。这里将梯子的数量变为 0，同时玩家只能用两种投掷骰子的手法：投掷 1~3 的投掷手法和投掷 1~6 的投掷手法。我们可以直接推理出最优的方案：

（1）在前进至 97、98、99 三个位置之前，全部使用投掷 1~6 的手法显然可以获得最优的前进步数。

（2）到了 97、98、99 这三个位置时，最好使用投掷 1~3 的手法，因为这样有更大的概率一次性到达终点。

由于这个策略比较简单，我们可以用一个数组表示，我们限定 Agent 只有 1~3 和 1~6 两种策略，于是可以构建以下三种策略。如代码所示：

```
# 最优策略
policy_ref = [1] * 97 + [0] * 3
# 全部投掷1-3的策略
policy_0 = [0] * 100
# 全部投掷1-6的策略
policy_1 = [1] * 100
```

代码中实现了三种策略：最优策略、全部执行第一个行动的策略和全部执行第二个行动的策略。为了简化代码，我们直接用一个数组表示 Agent 要执行的行动。完成了策略的构建，就可以评估策略的效果了，这其实就是让 Agent 在真实的环境上进行交互，得到回报总和。代码如下所示：

```
def eval_game(env, policy):
    state = env.reset()
    return_val = 0
    while True:
        if isinstance(policy, TableAgent):
            act = policy.play(state)
        elif isinstance(policy, list):
            act = policy[state]
```

```
        else:
            raise Error('Illegal policy')
        state, reward, terminate, _ = env.step(act)
        return_val += reward
        if terminate:
          break
    return return_val


def test_easy():
    sum_opt = 0
    sum_0 = 0
    sum_1 = 0
    env = SnakeEnv(0, [3, 6])
    for i in range(10000):
        sum_opt += eval_game(env, policy_ref)
        sum_0 += eval_game(env, policy_0)
        sum_1 += eval_game(env, policy_1)
    print 'opt avg={}'.format(sum_opt / 10000.0)
    print '0 avg={}'.format(sum_0 / 10000.0)
    print '1 avg={}'.format(sum_1 / 10000.0)
```

在代码中，我们使用每一种策略进行 1 万局游戏，并显示每一种策略的平均得分。由于棋盘上没有梯子，所以棋局的环境不用发生变化。游戏最终的平均得分如下：

```
0 avg=49
1 avg=68
opt avg=70
```

可以看出，经过设计的策略获得了最高的平均得分。下面我们就使用前面提到的思路，实现优化算法。这个方法分两步，首先是计算当前策略的值函数估计。

想要求解值函数，就必须用到 6.1 节提到的 Bellman 公式：

$$v(\boldsymbol{s}_t) = \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}_t) \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a})[\boldsymbol{r}_{t+1} + \gamma v(\boldsymbol{s}_{t+1})]$$

在 6.1 节我们曾提到，对于这个公式，我们利用了时间不变性这样的假设，于是等号左右两边的值函数值应该是相等的，所以这个问题可以采用解方程的方法求解。上

面的公式是采用连加的形式描述的，我们现在将其变成矩阵运算的形式，于是等式变成了图 6-3 所示的形式。

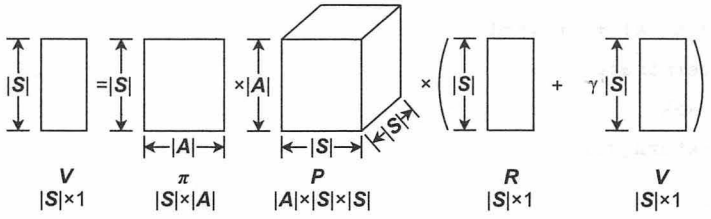


图 6-3　Bellman 公式的矩阵运算形式

如果用 $\boldsymbol{V}$ 表示状态值函数的向量，$\boldsymbol{\Pi}$ 表示策略矩阵，$\boldsymbol{P}$ 表示状态转移矩阵，$\boldsymbol{R}$ 表示回报向量，那么矩阵版的 Bellman 公式为

$$\boldsymbol{V} = \boldsymbol{\Pi}\boldsymbol{P}(\boldsymbol{R} + \gamma \boldsymbol{V})$$

通过求解可以得到

$$\boldsymbol{V} = (1 - \gamma \boldsymbol{\Pi}\boldsymbol{P})^{-1} \boldsymbol{\Pi}\boldsymbol{P}\boldsymbol{R}$$

只要 $(1 - \gamma \boldsymbol{\Pi}\boldsymbol{P})^{-1}$ 的逆矩阵存在，状态价值就可以解出来。实际中由于矩阵数值的限制，逆矩阵是存在的，所以这种方法可以求解。但是对于一些状态和行动比较多的问题，采用这种方法求解的复杂度偏高，因此实际中大家不用这种方法，而是采用迭代的方法进行计算。

那么迭代的方法如何计算呢？在其他数值计算算法中，我们会利用旧参数迭代更新新参数的方法，例如梯度下降法，这里也将采用类似的方法。我们将上面的 Bellman 公式左右两边的值函数赋予不同的值，右边是当前值，左边是新迭代的值，假设迭代轮数为 $\{1, 2, 3, \ldots, T\}$，那么迭代更新的公式就变为

$$v^T(\boldsymbol{s}_t) = \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}_t) \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a})[\boldsymbol{r}_{t+1} + \gamma v^{T-1}(\boldsymbol{s}_{t+1})]$$

由于有 γ 的存在，每个状态的价值最终将得到收敛，由于采用了表格式的表示方法，每个状态的策略实际上是确定的，因此不必进行概率计算，直接根据行动进行转换即可。我们将迭代的版本实现出来，如下所示：

```
def policy_evaluation(self, agent, max_iter = -1):
    iteration = 0
    # iterative eval
```

```
    while True:
        # one iteration
        iteration += 1
        new_value_pi = agent.value_pi.copy()
        for i in range(1, agent.s_len): # for each state
            value_sas = []
            ac = agent.pi[i]
            for j in range(0, agent.act_num): # for each act
                transition = agent.p[ac, i, :]
                value_sa = np.dot(transition, agent.r + agent.gamma *
                agent.value_pi)
                value_sas.append(value_sa)
            new_value_pi[i] = value_sa# value_sas[agent.policy[i]]
        diff = np.sqrt(np.sum(np.power(agent.value_pi - new_value_pi,
            2)))
        if diff < 1e-6:
            break
        else:
            agent.value_pi = new_value_pi
        if iteration == max_iter:
            break
```

完成了这一步，下一步就是根据前面的状态值函数计算状态-行动值函数：

$$q(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)[\boldsymbol{r}_t + \gamma v(\boldsymbol{s}_{t+1})]$$

完成计算后，就可以根据同一状态下的行动价值更新策略：

$$\pi(\boldsymbol{s}) = \text{argmax}_{\boldsymbol{a}} q(\boldsymbol{s}, \boldsymbol{a})$$

这样就完成了状态的更新。代码如下所示：

```
def policy_improvement(self, agent):
    new_policy = np.zeros_like(agent.pi)
    for i in range(1, agent.s_len):
        for j in range(0, agent.a_len):
            agent.value_q[i,j] = np.dot(agent.p[j,i,:], agent.r + agent.
            gamma * agent.value_pi)
```

```
            # update policy
            max_act = np.argmax(agent.value_q[i,:])
            new_policy[i] = max_act
        if np.all(np.equal(new_policy, agent.pi)):
            return False
        else:
            agent.pi = new_policy
            return True
```

将上面的代码联起来，整个算法的执行如下所示：

```
def policy_iteration(self, agent):
    iteration = 0
    while True:
        iteration += 1
        self.policy_evaluation(agent)
        ret = self.policy_improvement(agent)
        if not ret:
            break
    print 'Iter {} rounds converge'.format(iteration)
```

那么这个算法最终执行的结果如何呢？

```
def policy_iteration_demo1():
    env = SnakeEnv(0, [3,6])
    agent = TableAgent(env)
    pi_algo = PolicyIteration()
    pi_algo.policy_iteration(agent)
    print 'return_pi={}'.format(eval_game(env, agent))
    print agent.pi
```

结果为：

```
Iter 2 rounds converge
return_pi=70
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

可以看出，算法求出的策略结果和我们设想的结果一样，这说明算法在这个例子上是没问题的，而这个算法就被称为策略迭代法。可以看出，每一轮迭代结束，策略都

进行了一次更新，当策略没有更新时，迭代结束。前面提到算法的两个部分也分别被称为：策略评估部分（Policy Evaluation）和策略提升部分（Policy Improvement）。

从上面的分析中我们发现，策略评估部分在逻辑上没有问题，值函数会随着迭代逐渐收敛，主要的问题出现在策略提升上，我们能不能证明这种改进方案一定得到最优结果呢？当然可以。如果把每一轮迭代生成的策略形成一个策略组，并以迭代轮数进行编号，那么可以得到一个策略列 $\{\pi_t\}$，我们不但可以证明随着 t 的增大，这个策略列依值函数最终收敛到最优的策略 π^*，也就是说

$$\lim_{t\to\infty} |v(\pi_t) - v(\pi^*)| = 0$$

而且可以证明这个策略列依值函数一致收敛到最优策略，即对于任意的数 ϵ，我们都存在一个 k，使得当 $t > k$ 时：

$$|v(\pi_t) - v(\pi^*)| \leqslant \epsilon$$

换句话说，随着迭代的进行，策略不断趋近于最优，每一轮迭代的策略都不会比前一轮迭代的策略差。6.2.2 节就来证明这一点。

6.2.2 策略提升的证明

我们利用 Bellman 公式证明。假设现在有一个策略 π，经过计算得到了对应的值函数 $v(\pi)$，并以此求出对应的 $q(\pi)$。这时我们发现存在一个状态 $\hat{\boldsymbol{s}}$ 和 $\hat{\boldsymbol{a}}$，使得

$$q(\hat{\boldsymbol{s}}, \hat{\boldsymbol{a}}) \geqslant q(\hat{\boldsymbol{s}}, \pi(\hat{\boldsymbol{s}}))$$

那么我们就可以对这部分策略进行更新，得到一个新的策略 π^+，由于采用“表格式”的表示形式，这个新策略除了在某个状态 $\hat{\boldsymbol{s}}$ 的决策与原策略不同，其他完全一致，它的形式可以写作：

$$\pi^+(\boldsymbol{s}) = \begin{cases} \pi(\boldsymbol{s}) & \boldsymbol{s} \neq \hat{\boldsymbol{s}} \\ \hat{\boldsymbol{a}} & \boldsymbol{s} = \hat{\boldsymbol{s}} \end{cases}$$

那么对于任意一个状态 $\boldsymbol{s}$，有

$$\begin{aligned} v_\pi(\boldsymbol{s}_t) &= q_\pi(\boldsymbol{s}_t, \pi(\boldsymbol{s}_t)) \\ &\leqslant q_\pi(\boldsymbol{s}_t, \pi^+(\boldsymbol{s}_t)) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{\boldsymbol{a}} \pi^{+}(\boldsymbol{a}|\boldsymbol{s}_t) \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a})[\boldsymbol{r}_{t+1} + \gamma v_{\pi}(\boldsymbol{s}_{t+1})] \\
&\leqslant \sum_{\boldsymbol{a}} \pi^{+}(\boldsymbol{a}|\boldsymbol{s}_t) \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a})[\boldsymbol{r}_{t+1} + \gamma q_{\pi}(\boldsymbol{s}_{t+1}, \pi^{+}(\boldsymbol{s}_{t+1}))] \\
&= E_{\pi^{+},p}[\boldsymbol{r}_{t+1} + \gamma q_{\pi}(\boldsymbol{s}_{t+1}, \pi^{+}(\boldsymbol{s}_{t+1}))] \\
&= E_{\pi^{+},p}[\boldsymbol{r}_{t+1} + \gamma E_{\pi^{+},p}[\boldsymbol{r}_{t+2} + \gamma v_{\pi}(\boldsymbol{s}_{t+2})]] \\
&= E_{\pi^{+},p}[\boldsymbol{r}_{t+1} + \gamma \boldsymbol{r}_{t+2} + \gamma^2 v_{\pi}(\boldsymbol{s}_{t+2})] \\
&\leqslant E_{\pi^{+},p}[\boldsymbol{r}_{t+1} + \gamma \boldsymbol{r}_{t+2} + \gamma^2 \boldsymbol{r}_{t+3} + \gamma^3 v_{\pi}(\boldsymbol{s}_{t+3})] \\
&\leqslant E_{\pi^{+},p}[\boldsymbol{r}_{t+1} + \gamma \boldsymbol{r}_{t+2} + \gamma^2 \boldsymbol{r}_{t+3} + \gamma^3 \boldsymbol{r}_{t+4} + \cdots] \\
&= v_{\pi^{+}}(\boldsymbol{s}_t)
\end{aligned}
$$

所以可以证明，每一次策略提升都不会使当前策略的价值下降，同理可以证明，如果策略 π_1 下状态的价值不高于策略 π_2 下状态的价值（$v_{\pi_1}(\boldsymbol{s}) \leqslant v_{\pi_2}(\boldsymbol{s})$，$\forall \boldsymbol{s}$），且 π_2 下状态的价值不高于 π_3（$v_{\pi_2}(\boldsymbol{s}) \leqslant v_{\pi_3}(\boldsymbol{s})$，$\forall \boldsymbol{s}$），那么 π_1 下状态的价值也不高于 π_3（$v_{\pi_1}(\boldsymbol{s}) \leqslant v_{\pi_3}(\boldsymbol{s})$，$\forall \boldsymbol{s}$），基于这种传递性，就可以得到策略迭代不断趋近最优的性质。

6.2.3 策略迭代的效果展示

前面证明了策略迭代的收敛性质，接下来介绍前面例子中的一些细节。假设一开始所有的策略都采用 1~3 的投掷手法，于是在第一轮策略评估中，共进行了 94 轮迭代，过程中的状态的迭代值在不断变化，我们以“50”这个位置为例，图 6-4 所示为 94 轮迭代下价值的变化值。

横轴为迭代轮数，纵轴为价值，可以看出随着迭代轮数的增加，价值总体趋于平稳。完成第一轮的策略提升后，实际上策略已经被更新为最优策略，于是在第二轮策略评估中，再经过 94 轮迭代，“50”位置的价值又经历了如下变化，如图 6-5 所示。

看完上面那个简单的例子后，我们回到复杂的例子中，对一个拥有 10 个梯子的问题，策略迭代会给我们什么答案呢？

```
def policy_iteration_demo2():
    env = SnakeEnv(10, [3,6])
    agent = TableAgent(env)
    agent.pi[:]=0
    print '0 avg={}'.format(eval_game(env,agent))
    agent.pi[:]=1
```

```
print '1 avg={}'.format(eval_game(env,agent))
agent.pi[97:100]=0
print 'opt avg={}'.format(eval_game(env,agent))
pi_algo = PolicyIteration()
pi_algo.policy_iteration(agent)
print 'pi avg={}'.format(eval_game(env,agent))
print agent.pi
```

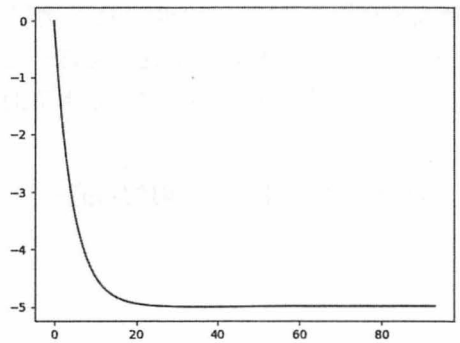


图 6-4 第一轮策略评估时位置“50”的价值变化图

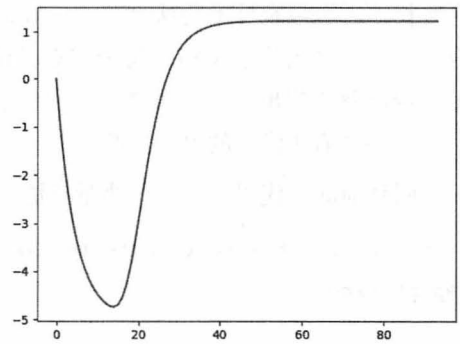


图 6-5 第二轮策略评估时位置“50”的价值变化图

结果如下：

```
0 avg=-45
1 avg=21
opt avg=31
pi avg=41
[0 1 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 0 1 0 1 1 1 1 1
 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0 0]
```

可以看出，策略迭代的方法优于前面的三种方法。可以看出它的策略中已经将两种手法混合使用了。笔者猜想它的策略思想如下：

（1）在靠近上升梯子附近使用 1~3 的投掷手法。

（2）在靠近下降梯子或者无梯子时使用 1~6 的投掷手法。

（3）最后几步使用 1~3 的投掷手法。

所以我们可以从最终的策略中猜出棋盘的样子及梯子所在的位置。以上就是策略迭代的算法，除了这种算法外，在接下来的章节我们还会介绍其他算法。

6.3 价值迭代

6.3.1 N 轮策略迭代

6.2 节介绍了策略迭代的算法，它可以解决蛇棋的策略规划问题，帮助我们更好地完成游戏。从算法中可以看出，算法的主要时间都花费在策略评估上，对一个简单的问题来说，在策略评估上花费的时间不算长；但对复杂的问题来说，这个步骤的时间实在有些长。一个最直接的想法就是——我们能不能缩短在策略评估上花的时间呢？例如，我们大概估计出当前策略下每个状态的值函数就好，虽然不够精确，但这样也许已经足以帮助我们找出最优的策略了，再做更精细的评估实际上并不必要。这就是本节的主角——价值迭代法的思想来源。

回到前面的代码中，为了测量算法的用时，我们将在代码中注入计时的功能：

```
from contextlib import contextmanager
import time

@contextmanager
def timer(name):
    start = time.time()
    yield
    end = time.time()
    print '{} COST:{}'.format(name, end - start)
```

然后将策略迭代的代码进行改进：

```
class PolicyIterationWithTimer(PolicyIteration):
    def policy_iteration(self, agent, max_iter=-1):
        iteration = 0
        while True:
            iteration += 1
            with timer('Timer PolicyEval'):
                self.policy_evaluation(agent, max_iter)
            with timer('Timer PolicyImprove'):
                ret = self.policy_improvement(agent)
            if not ret:
                break
```

对于有 10 个梯子的问题，我们固定随机数，代码将变为：

```
def policy_iteration_demo():
    env = SnakeEnv(10, [3,6])
    agent = TableAgent(env)
    pi_algo = PolicyIterationWithTimer()
    pi_algo.policy_iteration(agent)
    print 'return_pi={}'.format(eval_game(env,agent))
    print agent.pi
```

可以得到以下的时间消耗记录：

```
policy evaluation proceed 94 iters.
Timer PolicyEval COST:0.139311790466
Timer PolicyImprove COST:0.00118112564087
policy evaluation proceed 62 iters.
Timer PolicyEval COST:0.0768580436707
Timer PolicyImprove COST:0.000974178314209
policy evaluation proceed 46 iters.
Timer PolicyEval COST:0.0550677776337
Timer PolicyImprove COST:0.00197505950928
Iter 3 rounds converge
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

从时间记录可以看出，PolicyEval 的时间远大于 PolicyImprove，所以策略评估确实是最花时间的。那么我们就从这里进行改进，首先限制策略评估的迭代轮数，在上面的测试中，策略迭代进行了 94 轮，我们把它降至 50 轮可以吗？

将策略评估限制在 50 轮，得到如下输出（有省略）：

```
policy evaluation proceed 46 iters.
Timer PolicyEval COST:0.0480210781097
Timer PolicyImprove COST:0.00193190574646
Iter 3 rounds converge
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

继续激进一点，把迭代轮数改到 10：

```
policy evaluation proceed 11 iters.
Timer PolicyEval COST:0.011638879776
Timer PolicyImprove COST:0.00103783607483
Iter 4 rounds converge
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

那么剩下的就是最激进的，把迭代轮数改到 1：

```
policy evaluation proceed 2 iters.
Timer PolicyEval COST:0.00178599357605
Timer PolicyImprove COST:0.00106501579285
Iter 7 rounds converge
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

我们发现，随着策略评估的迭代轮数不断降低，算法的总迭代数在升高。当策略评估的迭代轮数为 94 时，算法总迭代数为 3；而策略评估的迭代轮数为 1 时，算法总迭代数变为 7，但是这样的改变并不会影响最终的结果，反而使它成了这些实验中最快的算法。这个算法就是评估迭代轮数为 1 的策略评估算法。

实际上，我们可以认为评估迭代轮数为 1 的策略迭代法很像基于价值的迭代法。为什么呢？在策略迭代算法中，每一轮迭代过后，状态价值得到了收敛，但是当前策略不一定是最优的，直到策略不再发生变化，训练才会结束。策略迭代的过程如图 6-6 所示，其中横轴表示值函数的收敛效果，数值到达 ∞ 时完成收敛，纵轴表示策略的优异度，数值到达 ∞ 时策略到达最优。

而对于 1 轮迭代来说，在策略达到最优前，值函数都不会收敛。如果值函数收敛，则说明值函数没有发生改变，策略也就随之收敛了。对应的迭代过程图如图 6-7 所示。

从图 6-7 所示的更新路线可以看出，与其说这是更新策略，不如说是更新价值。当价值迭代完成后，策略迭代也就随之完成，所以我们可以将迭代的重点放在价值上。6.3.2 节将介绍真正的价值迭代法。

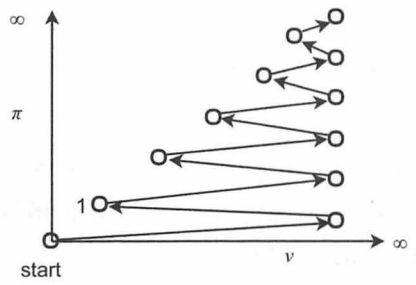


图 6-6 策略迭代算法的更新路线

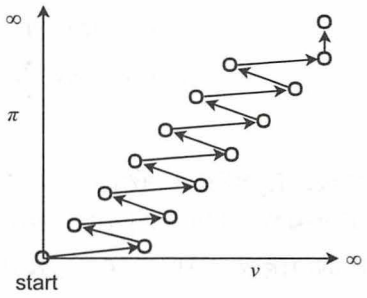


图 6-7 价值迭代算法的更新路线

6.3.2 从动态规划的角度谈价值迭代

从前面的介绍可以看出，策略迭代算法相对容易理解，因为它的思路很清晰，而价值迭代算法看上去似乎是一个“贪心”版的策略迭代算法，它的收敛性质能否保证呢？回答这个问题需要从“贪心”的对立面——动态规划解释。

前面我们一直站在策略的角度分析，并且把价值迭代法定义为一轮迭代的策略迭代法，那么对于一轮迭代的算法，它的形式能否做一个改变呢？我们知道策略评估这一步完成了下面的计算：

$$v(\boldsymbol{s}_t) = \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}_t) \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a})[\boldsymbol{r}_{t+1} + \gamma v(\boldsymbol{s}_{t+1})]$$

接下来的策略改进完成了策略改进的计算：

$$q(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)[\boldsymbol{r}_{t+1} + \gamma v(\boldsymbol{s}_{t+1})]$$

$$\pi(\boldsymbol{s}) = \text{argmax}_{\boldsymbol{a}} q(\boldsymbol{s}, \boldsymbol{a})$$

能不能将这三个步骤结合，成为一个关于值函数的更新步骤呢？首先将后两个公式合并，得到

$$\pi(\boldsymbol{s}) = \text{argmax}_{\boldsymbol{a}} \sum_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)[\boldsymbol{r}_{t+1} + \gamma v(\boldsymbol{s}_{t+1})]$$

由于最优策略的存在，实际上策略最终的选择是单一的。也就是说，对于每一个状态，最优策略会采取某一种行动，这种行动不会比其他行动差，所以我们可以得到状态值函数的更新方法：

$$v(s) \leftarrow \max_a q(s,a)$$
$$v(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \tilde{v}(s')]$$

经过这样的转换，我们终于将视角转换到值函数上。可以看到公式的左右两边都出现了值函数，其中等号左边的值函数是迭代更新后的值函数，等号右边的值函数是上一轮的值函数。这时，这个公式就展示出动态规划的“气质”。

相信很多读者在学习算法时都接触过动态规划相关的问题。算法的一大核心思想是分解问题，将大问题划分成一个个小问题，然后用小规模下的解决方案解决小规模的问题。动态规划的思路也基本相同，它有两个核心特点：最优子结构和重复子结构。

所谓**最优子结构**，是指一个子问题的最优解是可以得到的。对应蛇棋的问题，可以理解为是“从某个位置出发行走一步能够获得的最大奖励”的问题，由于只走一步，这个问题很容易计算。

所谓**重复子结构**，是指一个更大的问题是由一些小问题组成的，而求解不同的大问题时可能会用上同一个子问题，子问题被重复利用，计算量也就减少了。对应蛇棋的问题，可以理解为是“从某个位置出发行走两步能够获得的最大奖励”的大问题，利用前面已经得到的子问题，这个大问题可以用伪代码表示：

```
“某个位置走两步的最大奖励” =max([这一步的奖励 + 从到达位置出发走一步获得
的最大奖励 for 走法 in 可能的走法])
```

可以看出，这个公式用到了最优子结构的性质。由于打折率的存在，值函数的绝对值存在上界，值函数在这样的更新中最终会得到收敛，所以这个方法是可行的。

想要理解价值迭代和动态规划的关系，除了使用公式表示外，还可以使用图像。这里需要对问题做一个假设：假设所有的问题都存在一个“终点时刻”，对有明确终止条件的问题来说，这样的终点当然存在；但对没有终点状态的问题来说，可以假设某一个时刻为终点，例如时间的终点、外物灭亡时刻的终点……那么我们可以记录这一时刻为 T，所有状态的价值为 0，由于没有后面的行动，此刻价值设为 0 也是正常的。T 时刻的状态与价值情况如图 6-8 所示。

知道了时刻 T 的情况，我们就能得到 $T-1$ 时刻从每个状态出发能获得的最优价值。对于每一个状态，我们遍历所有的行动，通过 Bellman 公式，可以得到这一时刻的最优长期回报，如图 6-9 所示。

再向前一步，来到 $T-2$ 时刻，我们同样可以利用 $T-1$ 时刻的结果计算出 $T-2$ 时刻的最优价值。依此类推，我们可以向前推进无数步，直到价值收敛，此时的收敛值就是从当前位置到时间终结点的最优价值，也就是价值迭代法要求出的结果。

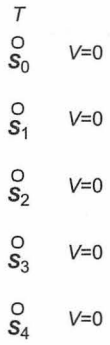


图 6-8 T 时刻的状态与价值情况

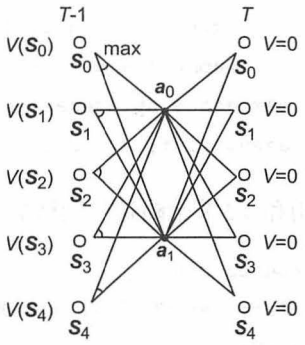


图 6-9 $T-1$ 时刻的状态与价值情况

6.3.3 价值迭代的实现

根据前面的介绍，我们可以给出价值迭代的代码，代码如下所示：

```
def value_iteration(self, agent, max_iter = -1):
    iteration = 0
    while True:
        iteration += 1
        new_value_pi = np.zeros_like(agent.value_pi)
        for i in range(1, agent.s_len): # for each state
            value_sas = []
            for j in range(0, agent.a_len): # for each act
                value_sa = np.dot(agent.p[j, i, :], agent.r + agent.gamma
                    * agent.value_pi)
                    value_sas.append(value_sa)
            new_value_pi[i] = max(value_sas)
        diff = np.sqrt(np.sum(np.power(agent.value_pi - new_value_pi, 2)))
        if diff < 1e-6:
            break
        else:
            agent.value_pi = new_value_pi
        if iteration == max_iter:
            break
    print 'Iter {} rounds converge'.format(iteration)
    for i in range(1, agent.s_len):
        for j in range(0, agent.a_len):
```

```
            agent.value_q[i,j] = np.dot(agent.p[j,i,:], agent.r + agent.
            gamma * agent.value_pi)
        max_act = np.argmax(agent.value_q[i,:])
        agent.pi[i] = max_act
```

使用和 6.2 节策略迭代同样的实验，也能得到同样的结果，那么它的用时是多少呢？

```
Iter 3 rounds converge
Timer PolicyIter COST:0.190360069275
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
Iter 94 rounds converge
Timer ValueIter COST:0.0884821414948
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

可以看出，价值迭代和策略迭代得到了相同的结果，但它比策略迭代快一些。这个结果和两个算法在其他问题上的效果一致。读者可以看出，价值迭代的速度没有本节中的一轮策略评估迭代速度快，一轮策略评估迭代的时间为 0.023s。这说明两个算法仍然存在一定的区别，所以笔者提到两者的思想相近，但并不相同。那么它们的区别是什么呢？我们还能不能让训练速度更快呢？

6.4 泛化迭代

6.4.1 两个极端

在 6.3 节我们留了一个悬念：为什么价值迭代法的运算速度不如一轮策略迭代法的速度快呢？本节将解答这个问题。在解答问题前，我们先对比策略迭代法和价值迭代法的特点。

策略迭代法的中心是策略函数，它通过反复执行“策略评估 + 策略提升”两个步骤使策略变得越来越好；价值迭代法的中心是值函数，它通过利用动态规划的方法迭代更新值函数，并最终求出策略函数。由此可以看出两者有如下特点。

（1）两个方法最终都求出策略函数和值函数。

（2）最优的策略函数都是由收敛的值函数得到的。

（3）值函数通过 Bellman 公式收敛。

由此发现一个关键：两者都需要训练和更新策略函数和值函数，只是侧重点不同。策略迭代的核心是策略，为了提升策略，值函数可以求解得准确，也可以求解得不那么准确；价值迭代的核心是价值，算法的核心部分根本没有出现与策略有关的内容，直到算法最后通过值函数求出策略。

两种方法都十分看重自己关心的那部分，可以选择忽略另一部分，因此可以看出两个方法都比较极端。既然找到了两个极端的方法，可不可以找到两种方法的中间地带呢？当然可以，这就是本节要介绍的广义策略迭代法（Generalized Policy Iteration）。

6.4.2 广义策略迭代法

在 6.3 节中，我们给出了策略迭代法和一轮策略评估迭代法的优化路径过程图。从图 6-6 和图 6-7 中可以看出，优化的起点在坐标轴的左下角，终点在右上角，于是读者可以想象，从左下角到右上角，一定有很多种走法，前面介绍的两种是其中的走法之二。我们可以再设计其他的更新算法，例如，我们将策略迭代和价值迭代融合起来，在优化的前几轮进行策略迭代，然后进行价值迭代，这样同样可以获得正确的结果，对应的优化过程如图 6-10 所示。

所谓的广义策略迭代法，就是定义一个迭代算法族，其中的算法都是由策略迭代和价值迭代算法组合而成的。组合的方法有很多，可以形成的方法也有很多，而前面提到的两种算法是广义策略迭代法的一种特例。由于其中的算法很多，这些算法中很有可能存在一些比前面两种算法速度更快的方法，例如前面多次提到的一轮策略评估迭代法。

了解了广义策略迭代法，再回到 6.3 节的结尾，分析为什么价值迭代的方法不是最快的。由于蛇棋问题的状态是离散的，对应的行动也是离散的，因此策略所在的空间也是离散的，这个问题的策略价值坐标轴实际上应该如图 6-11 所示。

由于策略空间是离散的，而值函数空间是连续的，同样的策略可能对应着一定范围内的值函数，因此最优策略也一定对应着一定范围的值函数。所以在使用价值迭代法时，需要将值函数更新到最优，所以为了使值函数收敛，它花费了一定的时间；而如果我们不去追求值函数的最优，当值函数接近最优时，使用它进行策略改进，也许就可以得到最优策略了，这样实际上可以节省一部分时间。

这样我们就可以根据蛇棋问题的特点，构造出对应的方法，使它的计算速度变得更快。

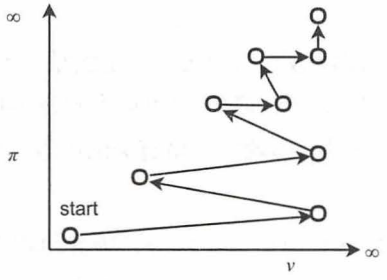


图 6-10　某种优化过程图

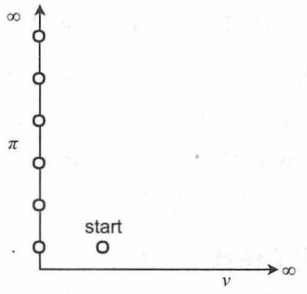


图 6-11　问题的真实空间图

6.4.3　泛化迭代的实现

泛化迭代是一个算法框架，想要实现好需要对框架做一定的设计。笔者简单地实现了其中的一个“计算较快的”算法，它直接将策略迭代和价值迭代结合。实现方法如下：

```
class GeneralizedPolicyIteration(object):
    def __init__(self):
        self.pi_algo = PolicyIterationWithTimer()
        self.vi_algo = ValueIteration()

    def generalized_policy_iteration(self, agent):
        self.vi_algo.value_iteration(agent, 10)
        self.pi_algo.policy_iteration(agent, 1)
```

这个函数先执行 10 轮价值迭代的优化，然后进行策略迭代，同时策略迭代中的策略评估只进行一个循环的更新。经过这样的设计，我们来看看它的用时：

```
env = SnakeEnv(10, [3,6])
agent = TableAgent(env)
pi_algo = PolicyIterationWithTimer()
with timer('Timer PolicyIter'):
    pi_algo.policy_iteration(agent)
print 'return_pi={}'.format(eval_game(env,agent))
pi_algo = ValueIteration()
with timer('Timer ValueIter'):
```

```
    pi_algo.value_iteration(agent)
print 'return_pi={}'.format(eval_game(env,agent))
pi_algo = GeneralizedPolicyIteration()
with timer('Timer GeneralizedIter'):
    pi_algo.generalized_policy_iteration(agent)
print 'return_pi={}'.format(eval_game(env,agent))
```

从代码中可以看出，同时进行了 3 组实验，分别是策略迭代、价值迭代和我们给出的一种广义策略迭代法，它们的结果如下：

```
Timer PolicyIter COST:0.186598777771
return_pi=84
=================
Timer ValueIter COST:0.0881521701813
return_pi=84
=================
Timer GeneralizedIter COST:0.0130021572113
return_pi=84
```

虽然实验比较粗糙，但还是可以明显看出几个算法在时间上的差距。通过一定的设计，泛化迭代确实可以做到更快的计算和收敛。

6.5 总结

本章介绍了 MDP、策略迭代法、价值迭代法和广义策略迭代法相关的知识，让我们一起回顾其中的要点。

（1）MDP 包含了状态与行动交替的序列，我们可以将很多问题转化成这样的交互形式。

（2）值函数表示了长期回报的期望值。常见的有状态值函数 $V(\boldsymbol{s})$ 和状态–行动值函数 $q(\boldsymbol{s},\boldsymbol{a})$。

（3）策略迭代法通过交替执行策略评估和策略提升两个步骤完成最优策略的学习。

（4）价值迭代法通过迭代更新值函数使值函数达到最优，然后更新策略函数。

（5）泛化迭代融合了两种迭代方式。

第二部分
最优价值算法

第二部分将介绍基于最优价值思想的算法，这类算法将重点放在值函数上，通过交互序列的信息学习价值模型，并通过价值模型更新策略，其中的思想和价值迭代法十分相似。随着强化学习和深度学习的共同发展，基于 Q-Learning 的算法获得了很大的突破，甚至达到了专家的水平，其中的代表为 Deep Q Network。笔者还会介绍很多基于 Deep Q Network 的改进算法，这些算法进一步提高了模型的能力。

第二部分共包含两章：

- 第 7 章将介绍 Q-Learning 算法的基础知识，并介绍 Deep Q Network 算法。
- 第 8 章将介绍基于 Deep Q Network 的改进算法，并介绍集改进算法于一身的 Rainbow 算法。

7 Q-Learning 基础

第 6 章介绍了强化学习的基础知识，学习了基于马尔可夫决策过程的强化学习框架，并学习了策略迭代、价值迭代和泛化迭代算法。我们曾提到一个很关键的前提条件：Agent 知道环境的状态转移概率。本章要把这个条件去掉，并介绍新的算法：Q-Learning 和基于深层模型的 Deep Q Network。

7.1 状态转移概率：从掌握到放弃

从本章起，我们开始解决更贴近实际的问题。前面提到我们接触过的问题有一个特点，即我们可以知道环境运转的细节，具体说就是知道状态转移概率。对蛇棋来说，我们可以看到蛇棋的棋盘，也就可以了解到整个游戏的全貌，当我们发现前方有一个梯子时，我们可以根据前方是上升梯子还是下降梯子来决定使用哪个骰子。这时我们相当于站在上帝视角，能够看清一切情况。于是，动态规划这样的算法成了我们的选择。

但一个可悲的事实是，在很多实际问题中，我们无法得到游戏的全貌，也就是说，状态转移的信息 $P(\boldsymbol{S}_{t+1}|\boldsymbol{S}_t, \boldsymbol{a}_t)$ 无法获得。一般来说，我们将知晓状态转移概率的问题称为“基于模型”的问题（**Model-based Problem**），将不知晓的称为“无模型”问题（**Model-free Problem**）。以下面的进阶版蛇棋为例，**在这个新版蛇棋中，我们将不再对玩家显示棋盘信息和骰子可能的投掷数目，每次当玩家选择完所使用的手法后，玩家将直接得到棋子的下一个位置，而不会知道其他任何信息。**

如果知道状态转移概率，我们就可以使用第 6 章介绍的学习算法，迭代执行策略评估和策略改进两个步骤得到最优策略。而对于“盲棋”，我们不知道状态转移概率，就

无法直接执行策略评估，即不能使用 Bellman 公式进行值函数的更新。为了解决这个问题，我们需要使用其他方法完成策略评估，这就要追溯回强化学习自身的特点：不断试错，也就是通过尝试与环境交互来解决策略评估的问题。实际上，这个不断尝试的思路和人类的一些学习方式比较接近，我们可以列出新算法的大体思路。

（1）确定一个初始策略（这和前面的算法一致）。

（2）用这个策略进行游戏，得到一些游戏序列（**Episode**）：$\{s_1, a_1, s_2, a_2, \ldots, s_n, a_n\}$。

（3）一旦游戏的轮数达到一定数目，就可以认为这些游戏序列代表了当前策略与环境交互的表现，就可以将这些序列聚合起来，得到状态对应的值函数。

（4）得到了值函数，就相当于完成了策略评估的过程，这样就可以继续按照策略迭代的方法进行策略改进的操作，得到更新后的策略。如果策略更新完成，则过程结束；否则回到步骤 2。

上面的流程十分清晰地介绍了学习的过程，此时学习的关键就落在了下面两个问题上：

（1）如何得到这些游戏序列。

（2）如何使用序列进行评估。

本章将重点解决这两个问题。接下来介绍两个算法：蒙特卡罗法（Monte Carlo Method）和时序差分法（Temporal Difference Method）。

7.2 蒙特卡罗方法

本节我们介绍蒙特卡罗法。在前面的章节里，我们曾介绍当环境信息，也就是状态转移概率已知时，可以使用 Bellman 公式，通过不断迭代得到状态–行动值函数：

$$q_\pi(s_t, a_t) = \sum_{s_{t+1} \in S} p(s_{t+1}|s_t, a_t)(r_{t+1} + v_\pi(s_{t+1}))$$

然后通过值函数进行策略改进。而在无模型问题中，状态转移概率将无法知晓。于是我们需要把公式转变为

$$q_\pi(s_t, a_t) = E_{s_{t+1} \sim p, \pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k}]$$

书籍是人类进步的阶梯

看到了等号右边的期望，我们很自然地联想到了蒙特卡罗法，它是一种通过随机采样估计期望值的方法，假设我们通过一些方法，从状态 s 和行动 a 开始不断地与环境交互，得到了大量的样本序列

$$\{s_t, a_t, s_{t+1}^i, a_{t+1}^i, s_{t+2}^i, a_{t+2}^i, \ldots\}_{i=1}^N$$

对应的回报序列为 $\{r_t, r_{t+1}^i, r_{t+2}^i, \ldots\}_{i=1}^N$

我们可以通过这些样本序列逼近真实的期望值，那么公式就可以近似等于：

$$q(s,a) \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{k=0}^{\infty}\gamma^k r_{t+k}$$

如果这个序列是有限的，我们也可以直接使用序列产生的回报和作为价值：

$$q(s,a) \simeq \frac{1}{N}\sum_{i=1}^{N}[r_1^i + r_2^i + \ldots]$$

这样就可以得出策略评估的结果，具体算法其实比较简单。

以上就是蒙特卡罗法的全部内容，我们可以将它的算法全过程总结如下。

（1）让 Agent 和环境交互后得到交互序列。

（2）通过序列计算出每一时刻的价值。

（3）将这些价值累积到值函数中进行更新。

（4）根据更新的值函数更新策略。

接下来让我们实现蒙特卡罗法的代码。同时，我们还要将这个算法的结果和策略迭代法进行比较，看看在不知道环境模型的情况下，蒙特卡罗法能否做到和策略迭代一样的效果。

完整的算法流程如下所示：

```
def monte_carlo_opt(self, agent, env):
    for i in range(10):
        for j in range(100):
            self.monte_carlo_eval(agent, env)
        self.policy_improve(agent)
```


书籍是人类进步的阶梯

其中包含了两个子算法。其中的 policy_improve 和前面的算法类似，都是完成下面公式的操作：

$$\pi(\boldsymbol{s}) = \text{argmax}_{\boldsymbol{a}} q(\boldsymbol{s}, \boldsymbol{a})$$

所以这里不再赘述，下面介绍 monte_carlo_eval 方法，这个方法又分成几个部分，首先要用当前的策略玩游戏，产生一系列的序列，并将这些序列保存起来：

```
state = env.reset()
episode = []
while True:
    ac = agent.play(state, self.epsilon)
    next_state, reward, terminate, _ = env.step(ac)
    episode.append((state, ac, reward))
    state = next_state
    if terminate:
        break
```

产生序列之后，我们可以沿时间的反方向计算每一个时刻的状态行动和对应的长期回报值：

```
value = []
return_val = 0
for item in reversed(episode):
    return_val = return_val * agent.gamma + item[2]
    value.append((item[0], item[1], return_val))
```

最后，我们将每一个状态–行动对应的长期回报值交给值函数进行更新：

```
# every visit
for item in reversed(value):
    agent.value_n[item[0]][item[1]] += 1
    agent.value_q[item[0]][item[1]] += (item[2] -  \
        agent.value_q[item[0]][item[1]]) /  \
        agent.value_n[item[0]][item[1]]
```

这里涉及一个小的改变，因为我们要计算期望价值，要将所有观测到的长期回报求平均值，那么令状态行动价值为 $\boldsymbol{q}$，当前的时间为 t，积累的数量为 N，我们要求的值为 $\boldsymbol{q}_t^N$，当前已知的值为 $\boldsymbol{q}_t^{N-1}$ 和 N，每一个时刻的价值为 $\hat{\boldsymbol{q}}_t^i$，于是可以得到

$$\boldsymbol{q}_t^N = \frac{1}{N}\sum_{i=0}^{N} \hat{\boldsymbol{q}}_t^i$$

书籍是人类进步的阶梯

$$
\begin{aligned}
&= \frac{1}{N}[\sum_{i=0}^{N-1} \hat{\boldsymbol{q}}_t^i + \hat{\boldsymbol{q}}_t^N] \\
&= \frac{1}{N} \times (N-1)\boldsymbol{q}_t^{N-1} + \frac{1}{N}\hat{\boldsymbol{q}}_t^N \\
&= \boldsymbol{q}_t^{N-1} + \frac{1}{N}(\hat{\boldsymbol{q}}_t^N - \boldsymbol{q}_t^{N-1})
\end{aligned}
$$

这样每一时刻我们都可以求出当前所有观测值的平均数，而且这个公式和常见的梯度下降法的更新公式

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\boldsymbol{\theta}} J$$

也十分相近，其中的 $\frac{1}{N}$ 等同于学习率 α，而 $\hat{\boldsymbol{q}}_t^N - \boldsymbol{q}_t^{N-1}$ 等同于目标函数的梯度 $\nabla_{\boldsymbol{\theta}} J$。当然，这样的相似并不是偶然的，在后面的章节我们还会介绍类似形式的公式。

上面的代码中我们实现了一次游戏的交互，下面来比较蒙特卡罗法和策略迭代法两种算法的优劣。代码如下所示：

```
def monte_carlo_demo():
    env = SnakeEnv(10, [3,6])
    agent = ModelFreeAgent(env)
    mc = MonteCarlo()
    with timer('Timer Monte Carlo Iter'):
        mc.monte_carlo_opt(agent, env)
    print 'return_pi={}'.format(eval_game(env,agent))
    print agent.pi

    pi_algo = PolicyIteration()
    with timer('Timer PolicyIter'):
        pi_algo.policy_iteration(agent)
    print 'return_pi={}'.format(eval_game(env,agent))
    print agent.pi
```

最终的结果为：

```
Timer Monte Carlo Iter COST:0.128234863281
return_pi=81
[0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1
 1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
policy evaluation proceed 94 iters.
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
policy evaluation proceed 62 iters.
policy evaluation proceed 46 iters.
Iter 3 rounds converge
Timer PolicyIter COST:0.329040050507
return_pi=84
[0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

可以看出，蒙特卡罗的效果比策略迭代差一些，由于使用蒙特卡罗法时我们对状态转移是未知的，因此存在一些差距是可以理解的。那么除此之外还有没有别的差距呢？下面就来分析其他的原因。

7.3 探索与利用

7.2 节介绍了如何使用交互得到的序列进行价值估计，本节就来讨论模型未知情况下的另一个问题，也是前面提到的关键问题之一：如何得到这些游戏序列。

当模型已知时，我们可以根据状态转移直接计算得到收敛的值函数，但当模型未知时，我们只能通过与环境交互得到交互序列。这里就存在一个问题：交互序列可以真实反映状态转移概率吗？在一些极端条件下这是可能的。我们假设通过与环境交互，所有的状态和行动组合全部被 Agent 经历过，而且每种情况都经历了足够多次，那么通过统计计算这些序列的长期回报，可以得到接近状态转移概率已知时得到的值函数。

但是对常见的问题来说，实现这个目标比较困难。主要原因在于可行的状态行动序列太多，想要对其进行一一尝试不太现实。虽然蛇棋的状态空间比较少，但棋盘中还是可能存在一些位置，这些位置几乎没有被玩家走到过，不是在此之前就遇到了梯子，就是投掷时刚好错过。由于缺乏数据，这一位置的长期回报无法精确统计。

对蛇棋这样的问题，我们还有一些更“暴力”的方法。因为状态是有限可数的，行动也是有限可数的，所以所有的状态-行动对是可以模拟出来的，我们可以通过设定不同的游戏起始状态，使玩家将所有的位置经历一遍。这样数据缺乏的问题就得到了比较好的解决。既然蛇棋共有 99 个可行位置，那么我们让 Agent 依次从这些位置出发，经过反复的尝试，就可以收集到所有状态行动的交互信息。

不幸的是，这样的好事并不存在于每一个问题上，还是有很多问题，它们的状态是近乎无限的。以 Atari 游戏为例，它的游戏界面是指定尺寸的图像，但并不是所有指定尺寸的图像都是 Atari 的游戏。因此，我们可以把所有 Atari 游戏画面所在的空间想象

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

成指定尺寸内所有图像空间的一个子空间，更麻烦的是，这个子空间的概率分布很难得到，因此遍历所有可能的画面是不可行的，我们必须采取其他方法。

那么，为了达到和基于模型的算法接近的效果，我们首先要做的是确保当前的问题有遍历所有状态-行动对的可能。一般来说，我们认为问题的开始状态是一个集合，我们从中任选一个状态开始，并沿着这个状态序列执行所有可行的行动，那么当这个流程被反复执行到一定次数时，问题的所有状态-行动对一定都会被经历。这段话听上去很拗口，又像是一段废话，但这是解决问题的基本条件。如果由于观察信息的损失的状态，导致无模型算法存在无法被感知和经历，那么无模型算法就很难得到最优解了。

既然我们认为这个前提一定满足，那么只要尽可能地让每一个状态下的行动都有概率执行，是不是就可以弥补模型未知的不足呢？这里可以做一个尝试，为了“雨露均沾”，我们必须让其他没有被选为最优策略的行动也参与到与环境交互的过程中，这样才能让每一个 $q(s,a)$ 都有值，这样做策略改进也就更有意义。

基于这个想法，我们改进了策略模块，这里采用一种叫 $\epsilon-$greedy 的算法，首先随机生成一个 0~1 的数，然后用这个随机数进行判断，如果随机数小于某个值 ϵ，就采用完全随机的方式产生行动，此时每个行动产生的概率是一样的；如果随机数不小于某个值 ϵ，就选择当前的最优策略。可以想象，在使用了这个算法后，很多由于策略而无法到达的状态也可以到达了，这样对状态价值的估计也会更准确，对应的示意图如图 7-1 所示。

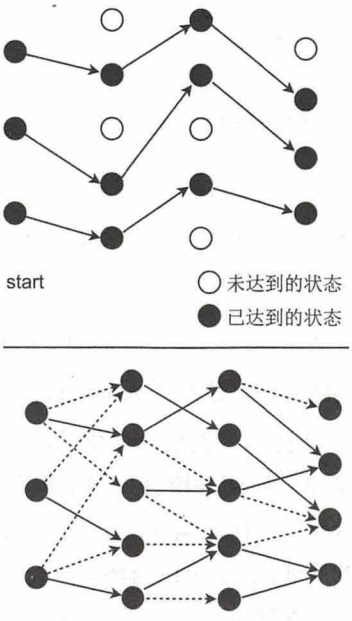


图 7-1 $\epsilon-$greedy 效果对比图

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

图 7-1 中上图所示为没有使用 $\epsilon-$greedy 的交互情况。图中的圆圈表示状态，因策略造成的状态转移用实线箭头表示。如果直接采用当前策略进行交互，由于策略是确定的，很可能有一些状态无法到达。使用 $\epsilon-$greedy 算法后，所有的行动都有可能被执行，这样可到达的状态也开始变多。图 7-1 中下图所示为使用 $\epsilon-$greedy 的交互情况，其中虚线箭头表示因随机策略造成的状态转移。$\epsilon-$greedy 的代码如下所示：

```
def play(self, state, epsilon = 0):
    if np.random.rand() < epsilon:
        return np.random.randint(self.a_len)
    else:
        return self.pi[state]
```

在这里，我们设定的 $\epsilon = 0.05$，完成了这一步的修改，我们再将算法和策略迭代法进行对比，结果如下所示：

```
Timer Monte Carlo Iter COST:0.486936807632
return_pi=84
[0 1 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 1 1 0 1 1 0 0
 0 0 1 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 1 0 1 1 1 0 1 1 0 1 1 1 0 0 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0]
policy evaluation proceed 94 iters.
policy evaluation proceed 62 iters.
policy evaluation proceed 46 iters.
Iter 3 rounds converge
Timer PolicyIter COST:0.325078964233
return_pi=84
[0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

可以看出，虽然两种方法的最终策略有部分不同，但是模拟得到的分数是相同的。说明在增加了对不同方法的尝试后，算法有了大幅的提高，达到了模型已知的水平。

前面提到的 $\epsilon-$greedy 算法实际上是在解决强化学习中的一个经典问题：**探索与利用**。这是两种与环境交互的策略。所谓的**探索**是指不拘泥于当前的表现，选择一些不同于当前策略的行动；**利用**就是持续使用当前的最优策略，尽可能地获得更多的回报。我们假设总的资源有限，例如时间有限，可以与环境进行交互的轮数有限，不能无止境地探索。所以，想要像前面所说等到所有的状态–行动对被多次经历是很难实现的。那么，如何在有限的时间达到最佳的效果呢？

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

如果只探索不利用，对每一个行动分配等量的时间，那么最优的策略可能没有经过充分的尝试，得到的值函数不够准确，会使玩家最终丧失选择它的机会，而对非最优行动花费了大量的时间，实际上也是一种浪费。如果只利用不探索，就容易满足于当前得到的行动，陷入次优解，与最优解擦肩而过。所以单独执行其中任意一种策略都不是最好的，必须同时执行两种策略才能达到良好的结果。上面介绍的 $\epsilon-$greedy 就是这样一种方法，它比较好地在两种策略间找到了平衡，用较高的性价比完成了交互的过程，并可以使用它求解出接近真实情况的价值。

7.4 蒙特卡罗的方差问题

7.2 节和 7.3 节介绍了蒙特卡罗法，它可以比较好地解决无模型场景下的蛇棋问题，在配合了 $\epsilon-$greedy 策略后，它的效果和策略迭代法十分相近。当然，它也不是完美的，这些不完美体现在很多方面，接下来我们就来看其中的一个方面：估计值的方差较大。

前面我们已经知道，蒙特卡罗通过与环境交互得到序列的回报信息，然后用这些回报信息求平均，就可以得到估计的值函数。这个方法从理论上看是没有问题的，它能够成立的根本原因在于概率论中的大数定理。大数定理也是没有问题的，可是大数定理只说明了统计量的收敛性，并没有说明收敛的速度。很显然，如果采样集合的方差比较大，那么想让均值收敛就需要更长的时间；如果采样序列的方差较小，那么收敛的速度也会相应地加快。

基于上面的分析，我们就来看看 7.3 节中蒙特卡罗法的采样存不存在方差方面的问题。我们用蒙特卡罗法完成一次计算，同时保存所有状态值为 50，行动值为 1 的长期回报值，并最终将所有的值综合起来做成一个直方图，如图 7-2 所示。

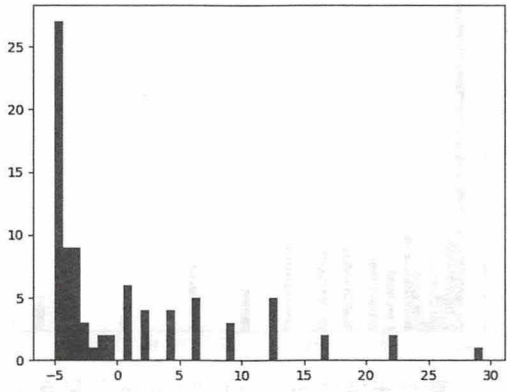


图 7-2 蒙特卡罗法 every-visit 方法的长期回报直方图

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

可以看出，由于持续时间比较长，长期回报的方差比较大，模型在收敛方面相对困难。导致方差大的因素有很多，例如每一次投掷骰子的数字都不同，那么到达终点的时刻也不一样，同一个位置计算得到的长期回报也自然不一样。

除此之外，就是交互序列中的采样频率这个因素。7.3 节的采样方法一般被称为 **every-visit**，也就是说，交互序列中的每一个状态-行动对都会参与到计算中，这样就为统计带来了困难。以蛇棋游戏为例，由于有梯子的存在，交互序列中可能会多次出现同一个状态，例如例子中的位置“50”。当采用 every-visit 方法时，同一状态的长期回报都会放入值函数中进行计算，在计算的过程中没有区分第一次到达“50”和第二次到达“50”的值，而是将这两个长期回报等权重地加在一起。由于这两次的长期回报存在差异，因此这样的统计方式会让方差增大，从而使期望值难以收敛。上述形式如图 7-3 所示。

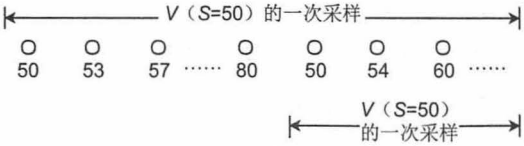


图 7-3　使用 every-visit 方法对重复出现的状态的处理

那么该如何应对这个问题呢？一个解决方案是把 every-visit 方法换掉，改成 first-visit 方法。所谓 first-visit 法就是只统计状态第一次出现时的长期回报，后面出现的同一状态则不再统计。从方法的理念来说，因为它只在一个序列中对相同状态取出一次，所以同一个交互序列内因多次出现同一状态而造成方差增大的问题将得到解决。我们将 every-visit 换成 first-visit，执行与上面相同的实验，可以得到如图 7-4 所示的直方图。

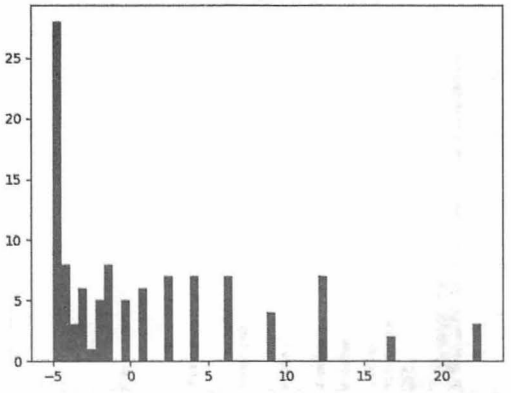


图 7-4　蒙特卡罗法 first-visit 的 return 收集直方图

从结果来看，方法 first-visit 的效果与 every-visit 相比稍好，但并没有明显的提高。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

因为前面提到的其他因素依然影响着结果，每次交互的数值差距依旧很大，这是蒙特卡罗法的缺点。但是在数据量足够大时，我们也可以看到它的优点：对期望值的估计是无偏的，这一点可以很轻易地证明。所以最终我们可以得到：蒙特卡罗法在估计价值上具有无偏差但方差较高的特点。当交互的数据量足够大时，蒙特卡罗法是有优势的，但是当交互的数据量不足时，它也会暴露出自己的缺点。

熟悉机器学习的读者一定会想到，这似乎和机器学习中的偏差与方差问题（bias and variance problem）类似，那么我们能不能找到一个低方差的方法呢？

7.5 时序差分法与 SARSA

为了使价值评估的方差进一步降低，本节将介绍另一种方法：时序差分法（Temporal Difference，简称 TD 法）。TD 法是一种结合了蒙特卡罗法和动态规划法的方法。从算法的主体结构来看，它同蒙特卡罗法类似，同样通过模拟交互序列的方式进行求解；但是从算法的核心思想来看，它同时用到了强化学习中的经典公式：Bellman 公式进行自迭代更新。

前面提到状态–行动值函数的 Bellman 公式的形式为

$$q_\pi(\boldsymbol{s},\boldsymbol{a})=\sum_{\boldsymbol{s}'}p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})[r(\boldsymbol{s}')+\sum_{\boldsymbol{a}'}\pi(\boldsymbol{a}'|\boldsymbol{s}')q(\boldsymbol{s}',\boldsymbol{a}')]$$

那么利用蒙特卡罗法，可以将公式变为

$$\begin{aligned}q(\boldsymbol{s},\boldsymbol{a})&=E_{(\boldsymbol{s},\boldsymbol{a})\sim\pi,p}[r(\boldsymbol{s}')+q(\boldsymbol{s}',\boldsymbol{a}')]\\&=\frac{1}{N}\sum_{i=1}^{N}[r(\boldsymbol{s}_i')+q(\boldsymbol{s}_i',\boldsymbol{a}_i')]\end{aligned}$$

TD 法和蒙特卡罗法有一些不同，蒙特卡罗法在估计价值时使用了完整序列的长期回报，而 TD 法使用了当前回报和下一时刻的价值估计，但是两个方法的公式在形式上还是有些类似的。如果把 TD 法像蒙特卡罗法一样写成迭代更新的形式，就可以得到

$$q_t(\boldsymbol{s},\boldsymbol{a})=q_{t-1}(\boldsymbol{s},\boldsymbol{a})+\frac{1}{N}[r(\boldsymbol{s}')+q(\boldsymbol{s}',\boldsymbol{a}')-q_{t-1}(\boldsymbol{s},\boldsymbol{a})]$$

这个公式就是 TD 法的基本形式。我们可以看出这个方法和蒙特卡罗法、动态规划法的关系。从形式上看，蒙特卡罗法会通过大量的采样估计出值函数的统计量，而 TD

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

法也同样需要采样这个过程；从更新公式上看，TD 法通过下一时刻的价值更新前一时刻的价值，似乎是承认了下一时刻的回报和价值足够优秀，从而利用了这个最优子结构进行更新。这和动态规划的思想是一致的。

前面叙述的算法属于 TD 法中的一种，被称为 SARSA 算法，这个名字看上去有些奇怪，它来自于这个方法的五个关键因子：S（当前状态）、A（当前行动）、R（模拟得到的奖励）、S（模拟进入的下一个状态）和 A（模拟中采取的下一个行动）。从上面的公式也可以看出，更新公式中确实只有这 5 个变量。蒙特卡罗法和 SARSA 算法的对比如图 7-5 所示。

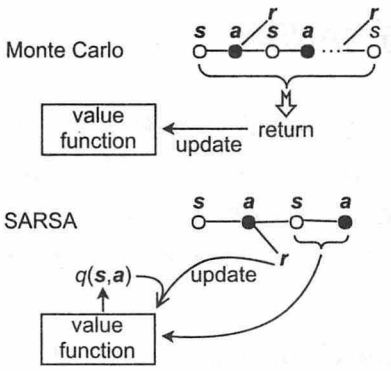


图 7-5　蒙特卡罗法和 SARSA 算法的对比

接下来就来实现这个算法，其实它的实现过程比较简单，这里只展现价值评估的过程，对应的代码和注释如下所示：

```
def sarsa_eval(self, env, agent):
    episode_num = 2000
    for i in range(episode_num):
        state = env.reset()
        prev_act = -1
        while True:
            act = agent.policy_act(state)
            state, reward, terminate, _ = env.step(act)
            if prev_act != -1:
                return_val = reward + (0 if state == -1 else agent.value_q
                    [state][act])
                agent.value_n[prev_state][prev_act] += 1
                agent.value_q[prev_state][prev_act] += (return_val - \
                    agent.value_q[prev_state][prev_act]) / \
```

仅供非商业用途或交流学习使用


```
                agent.value_n[prev_state][prev_act]
            prev_act = act
            prev_state = state

            if terminate:
                break
```

那么这个代码的实现效果如何呢？我们同样通过一段代码进行测试。

```
def td_demo():
    env = SnakeEnv(10, [3,6])
    agent = ModelFreeAgent(env)
    sarsa_algo = SARSA(0.05)
    with timer('Sarsa Iter'):
        sarsa_algo.sarsa(agent, env)
    print 'return_pi={}'.format(eval_game(env,agent))
    print agent.policy

    agent2 = TableAgent(env)
    pi_algo = PolicyIteration()
    with timer('Timer PolicyIter'):
        pi_algo.policy_iteration(agent2)
    print 'return_pi={}'.format(eval_game(env, agent2))
    print agent2.policy
```

为了试验效果，我们将 SARSA 算法中的迭代轮数增加到 2000，最终的结果为：

```
Timer Temporal Difference Iter COST:3.8632440567
return_pi=80
[0 0 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
policy evaluation proceed 94 iters.
policy evaluation proceed 62 iters.
policy evaluation proceed 46 iters.
Iter 3 rounds converge
Timer PolicyIter COST:0.330899000168
return_pi=84
[0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
```

```
0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0]
```

从结果可以看出，SARSA 算法的效果与蒙特卡罗法相比并不够好，那么它和蒙特卡罗法相比有什么优势呢？我们来思考前面提到的方差问题，我们同样取交互过程中状态为 50，行动值为 1 的价值估计值，再来看一看 SARSA 算法的结果，如图 7-6 所示。

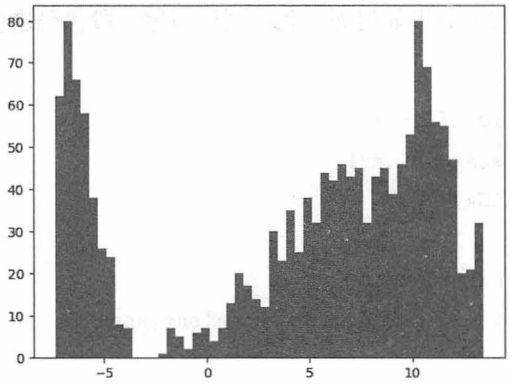


图 7-6　SARSA 算法的长期回报收集直方图

可以看出，SARSA 算法的数值变动和蒙特卡罗法相比更稳定，跨度比之前小。

那么，为什么 TD 系列的方法会在方差方面控制得更好呢？答案就在它的更新公式上。前面提到蒙特卡罗的计算方法由于使用了完整的采样得到了长期回报值，所以在价值的估计上的偏差更小，但同时它需要收集完整序列的信息，而序列存在一定的波动，所以价值的方差会比较大。

而 TD 方法只考虑了当前一步的回报值，其余的计算均使用了之前的估计值，所以当整体系统没有达到最优时，这样的估计都是存在偏差的，但是由于它只估计了一步，所以它在估计值方面受到的波动比较小，因此方差也会相应减小许多。

所以前人发现，蒙特卡罗法和 SARSA 算法象征着两个极端：一个为了追求极小的误差使方差变大，一个为缩小方差使误差变大。

7.6　Q-Learning

7.5 节我们介绍了基于 TD 思想的 SARSA 算法，它的核心公式为

$$q^T(\boldsymbol{s},\boldsymbol{a}) = q^{T-1}(\boldsymbol{s},\boldsymbol{a}) + \frac{1}{N}[r(\boldsymbol{s}') + q^{T-1}(\boldsymbol{s}',\boldsymbol{a}') - q^{T-1}(\boldsymbol{s},\boldsymbol{a})]$$

接下来，我们要介绍的另一种 TD 的算法叫作 Q-Learning，它的基本公式为

$$q^T(\boldsymbol{s},\boldsymbol{a}) = q^{T-1}(\boldsymbol{s},\boldsymbol{a}) + \frac{1}{N}[r(\boldsymbol{s}') + \max_{\boldsymbol{a}'} q^{T-1}(\boldsymbol{s}',\boldsymbol{a}') - q^{T-1}(\boldsymbol{s},\boldsymbol{a})]$$

两个算法的差别只在其中一项，SARSA 算法遵从了交互序列，根据下一步的真实行动进行价值估计；Q-Learning 算法没有遵循交互序列，而是在下一时刻选择了使价值最大的行动。这个差别使两种算法在效果上产生了一定的不同。我们先来看 Q-Learning 算法的效果，首先是算法的主体框架，这个框架和蒙特卡罗法、SARSA 算法基本一致，策略改进部分完全相同：

```
def q_learning(self, agent, env):
    iteration = 0
    while True:
        iteration += 1
        self.q_learn_eval(agent, env)
        ret = self.policy_improve(agent)
        if not ret:
            break
```

对应的策略评估代码为：

```
def q_learn_eval(self, agent, env):
    episode_num = 1000
    for i in range(episode_num):
        state = env.reset()
        prev_act = -1
        while True:
            act = agent.policy_act(state)
            state, reward, terminate, _ = env.action(act)
            if prev_act != -1:
                return_val = reward + (0 if state == -1 else np.max(agent.
                    value_q[state,:]))
                agent.value_n[prev_state][prev_act] += 1
                agent.value_q[prev_state][prev_act] += (return_val - \
                    agent.value_q[prev_state][prev_act]) / \
                    agent.value_n[prev_state][prev_act]
            prev_act = act
            prev_state = state
```

```
            if terminate:
                break
```

使用同前几节相同的评估方法，我们可以得到最终的评估结果：

```
Timer Temporal Difference Iter COST:4.24033594131
return_pi=81
[0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0]
policy evaluation proceed 94 iters.
policy evaluation proceed 62 iters.
policy evaluation proceed 46 iters.
Iter 3 rounds converge
Timer PolicyIter COST:0.318824052811
return_pi=84
[0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0]
```

可以看出，虽然 Q-Learning 算法和策略迭代法比还有点差距，但是 Q-Learning 算法比 SARSA 算法好一些。

前面提到了 SARSA 算法和 Q-Learning 算法在公式上的不同，实际上这两种算法代表了两种策略评估的方式，分别是 **On-Policy** 和 **Off-Policy**。On-Policy 对值函数的更新是完全依据交互序列进行的，我们在计算时认为价值可以直接使用采样的序列估计得到；Off-Policy 在更新值函数时并不完全遵循交互序列，而是选择来自其他策略的交互序列的子部分替换了原本的交互序列。从算法的思想上来说，Q-Learning 的思想更复杂，它结合了子部分的最优价值，更像是结合了价值迭代的更新算法，希望每一次都使用前面迭代积累的最优结果进行更新。

实际上，这也给 Q-Learning 算法带来了一些问题。由于它使用最优价值的行动替代了交互时使用的行动，所以在估计时就有可能造成对状态行动的“过高估计”。由于 Q-Learning 算法本质上还是采用了采样的方法，所以对于被采样的状态行动，Q-Learning 会对其“过高估计”，而对于没采样到的状态行动，则无法享受“过高估计”的待遇，那么两者之间就可能产生比较大的差距，这些差距会造成算法的波动，在真实的应用场景中更是如此。

7.7 Q-Learning 的收敛性分析

总的来说，Q-Learning 是一个比较优秀的算法，它具有较好的收敛性质。本节我们通过公式推导得出结果。

对于一般的收敛问题，我们通常需要一个框架对问题进行定义。一般认为问题定义的变量存在于一个空间中，空间拥有对于度量的定义，以及空间中某些运算的定义。同时，有的空间中存在一些“收缩算子”，对这些收缩算子来说，存在至少一个稳定点，使得我们从任意一个点出发，经过不断地与这些收缩算子运算，最终无限地靠近稳定点。如果我们可以证明一个算子是收缩算子，那么经过算子计算的点列就具备了一定的收敛性；如果稳定点恰好又具有我们想要的性质，那么这个算子就是我们想要的算子。上面的概念比较抽象，读者可能不太容易理解，下面我们把这些概念替换成 Q-Learning 中的概念。

- 前面提到 $q(\boldsymbol{s},\boldsymbol{a})$ 是状态行动值函数，我们定义一个空间，这个空间中的每一个点表示 $q(\boldsymbol{s},\boldsymbol{a})$ 的一种可能的形式。
- 度量可以衡量空间内两个点的距离，现在空间内的每一个点都是一个值函数，和我们常见的点并不相同，这里我们用无穷阶范数作为衡量两个“点”的距离的方法，范数的公式为

$$\|q_1 - q_2\|_\infty = \max_{\boldsymbol{s},\boldsymbol{a}} |q_1(\boldsymbol{s},\boldsymbol{a}) - q_2(\boldsymbol{s},\boldsymbol{a})|$$

  也就是说，找出两个值函数中相差最大的那一项作为距离。这个距离的定义十分重要，Q-Learning 算法并不是在所有的度量上都能收敛的。

- 空间中的算子 H 就是 Q-Learning 的计算公式：

$$(\mathrm{H}q)(\boldsymbol{s},\boldsymbol{a}) = \sum_{\boldsymbol{s}' \in \boldsymbol{S}} P_{\boldsymbol{a}}(\boldsymbol{s},\boldsymbol{s}')[r(\boldsymbol{s}) + \gamma \max_{\boldsymbol{a}} q(\boldsymbol{s}',\boldsymbol{a})]$$

  我们可以认为，通过计算，我们将空间中的一个“点”移动到了另一个“点”上。

- 空间中的稳定点就是最终收敛的值函数，我们记为 $q^*(\boldsymbol{s},\boldsymbol{a})$。
- 我们需要证明空间中的任意一点经过算子 H 的运算可以收敛到稳定点。

这个问题需要分两步证明，首先证明算子 H 是一个收缩算子（Contraction Operator），然后证明采用 Q-Learning 的计算方法可以收敛到最优价值。

我们先来证明第一步：对于算子 H，存在一个 $\gamma > 0$，使得对任意的值函数 q_1 和 q_2，都有

$$\|\mathrm{H}q_1 - \mathrm{H}q_2\|_\infty \leqslant \gamma\|q_1 - q_2\|_\infty$$

证明过程如下：

$$\begin{aligned}
&\|\mathrm{H}q_1 - \mathrm{H}q_2\|_\infty \\
&= \max_{s,a}|\sum_{s'\in S} P_a(s,s')[r(s) + \gamma\max_b q_1(s',b) - r(s) - \gamma\max_b q_2(s',b)]| \\
&= \max_{s,a}|\sum_{s'\in S} P_a(s,s')[\gamma\max_b q_1(s',b) - \gamma\max_b q_2(s',b)]| \\
&= \max_{s,a}\gamma|\sum_{s'\in S} P_a(s,s')[\max_b q_1(s',b) - \max_b q_2(s',b)]|
\end{aligned}$$

根据绝对值的不等式 $|a+b| \leqslant |a| + |b|$ 进行推导，可以得到

$$\begin{aligned}
&\leqslant \max_{s,a}\gamma\sum_{s'\in S}|P_a(s,s')(\max_b q_1(s',b) - \max_b q_2(s',b))| \\
&= \max_{s,a}\gamma\sum_{s'\in S} P_a(s,s')|\max_b q_1(s',b) - \max_b q_2(s',b)|
\end{aligned}$$

令 $s^*, b^* = \mathrm{argmax}_{s,b}|\max_b q_1(s,b) - \max_b q_2(s,b)|$，用它表示两个值函数差距的最大值。继续推导可以得到

$$\begin{aligned}
&\leqslant \max_{s,a}\gamma\sum_{s'\in S} P_a(s,s')|q_1(s^*,b^*) - q_2(s^*,b^*)| \\
&= \max_{s,a}\gamma|q_1(s^*,b^*) - q_2(s^*,b^*)| \\
&= \gamma|q_1(s^*,b^*) - q_2(s^*,b^*)| \\
&= \gamma\max_{s,b}|q_1(s,b) - q_2(s,b)| \\
&= \gamma\|q_1 - q_2\|_\infty
\end{aligned}$$

经过上面的证明，我们得出了想要的结果。下面就来证明第二步：Q-Learning 的收敛效果。

我们假设策略 π 能够覆盖所有的状态–行动组合：

$$P_\pi[A_t = a|S_t = s] > 0$$

用下面的公式对 q 函数进行更新：

$$q^{T+1}(\boldsymbol{x}_t, \boldsymbol{a}_t) = q^T(\boldsymbol{x}_t, \boldsymbol{a}_t) + \alpha^T(\boldsymbol{x}_t, \boldsymbol{a}_t)[\boldsymbol{r}_t + \gamma \max_{\boldsymbol{b}\in \boldsymbol{A}} Q_T(\boldsymbol{x}_{t+1}, \boldsymbol{b}) - Q_T(\boldsymbol{x}_t, \boldsymbol{a}_t)]$$

其中，$0 < \alpha_t(\boldsymbol{x}_t, \boldsymbol{a}_t) < 1$

在证明之前，我们需要使用一个定理：

输出在 $\mathbf{R}^n$ 上的随机过程 $\{\Delta_t\}$ 的更新过程定义为

$$\Delta_{t+1}(\boldsymbol{x}) = (1 - \alpha_t(\boldsymbol{x}))\Delta_t(\boldsymbol{x}) + \alpha_t(\boldsymbol{x})F_t(\boldsymbol{x})$$

如果下面的条件能够满足，那么它将依概率 1 收敛到 0：

（1）$0 \leqslant \alpha_t \leqslant 1, \sum_t \alpha_t(\boldsymbol{x}) = \infty$，同时保证 $\sum_t \alpha_t^2 < \infty$。

（2）$\|E[F_t(\boldsymbol{x})|F_t]\|_W \leqslant \gamma\|\Delta_t\|_W$，同时 $\gamma < 1$。

（3）$\text{var}[F_t(\boldsymbol{x})|F_t] \leqslant C(1 + \|\Delta_t\|_W^2)$，对所有的 $C > 0$。

如果我们能按照上面的定理整理出相应的形式，就能使用它得出 Q-Learning 算法收敛的证明。证明过程如下：

将 Q-Learning 的更新公式进行重新组合，可以得到

$$q^{T+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) = (1 - \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t))q^T(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t)[\boldsymbol{r}_t + \gamma \max_{\boldsymbol{b}\in \boldsymbol{A}} q^T(\boldsymbol{s}_{t+1}, \boldsymbol{b})]$$

假设值函数可以收敛且最终的收敛值为 $q^*(\boldsymbol{s}_t, \boldsymbol{a}_t)$，公式两边同时减去收敛值，可以得到

$$\begin{aligned} q^{T+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) - q^*(\boldsymbol{s}_t, \boldsymbol{a}_t) = {} & (1 - \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t))(q^T(\boldsymbol{s}_t, \boldsymbol{a}_t) - q^*(\boldsymbol{s}_t, \boldsymbol{a}_t)) \\ & + \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t)[r_t + \gamma \max_{\boldsymbol{b}\in \boldsymbol{A}} q^T(\boldsymbol{s}_{t+1}, \boldsymbol{b}) - q^*(\boldsymbol{s}_t, \boldsymbol{a}_t)] \end{aligned}$$

令 $\Delta^T(\boldsymbol{s}, \boldsymbol{a}) = q^T(\boldsymbol{s}, \boldsymbol{a}) - q^*(\boldsymbol{s}, \boldsymbol{a})$，可以得到

$$\begin{aligned} \Delta^{T+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) = {} & (1 - \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t))\Delta^T(\boldsymbol{s}_t, \boldsymbol{a}_t) \\ & + \alpha_t^T(\boldsymbol{s}_t, \boldsymbol{a}_t)[\boldsymbol{r}_t + \gamma \max_{\boldsymbol{b}\in \boldsymbol{A}} q_t^T(\boldsymbol{s}_{t+1}, \boldsymbol{b}) - q^*(\boldsymbol{s}_t, \boldsymbol{a}_t)] \end{aligned}$$

再令 $F^T(\boldsymbol{s}, \boldsymbol{a}) = r(\boldsymbol{s}, \boldsymbol{a}, X(\boldsymbol{s}, \boldsymbol{a})) + \gamma \max_{\boldsymbol{b}\in \boldsymbol{A}} q^T(\boldsymbol{s}_{t+1}, \boldsymbol{b}) - q^*(\boldsymbol{s}_t, \boldsymbol{a}_t)$

这样上面定理中随机过程的更新公式就可以得到了

$$\Delta^{T+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) = (1 - \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t))\Delta^T(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha^T(\boldsymbol{s}_t, \boldsymbol{a}_t)F^T(\boldsymbol{s}, \boldsymbol{a})$$

下面分别看定理需要满足的三个条件。其中条件一比较简单，我们假设

$$\alpha^T(\boldsymbol{x}_t, \boldsymbol{a}_t) = \frac{1}{T}$$

那么有

$$\sum_T \alpha^T > \int_1^\infty \frac{1}{T+1} \mathrm{d}T = \log(T+1)|_1^\infty = \infty$$
$$\sum_T (\alpha^T)^2 < 1 + \int_1^\infty \frac{1}{T^2} \mathrm{d}T = 1 + [-\frac{1}{T}]|_1^\infty \simeq 2 < \infty$$

这样就得到了证明。与此类似的数列还有很多，它们都适用于这个定理，而且在实践中也具有一定的操作性，这里就不再一一介绍。

对于条件二，我们有

$$\begin{aligned} E[F^T(\boldsymbol{x}, \boldsymbol{a}) | F^T] &= \sum_{\boldsymbol{y} \in \boldsymbol{\mathcal{X}}} P_{\boldsymbol{a}}(\boldsymbol{x}, \boldsymbol{y}) [r(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y}) + \gamma \max_{\boldsymbol{b} \in \boldsymbol{A}} q^T(\boldsymbol{y}, \boldsymbol{b}) - q^*(\boldsymbol{x}, \boldsymbol{a})] \\ &= \sum_{\boldsymbol{y} \in \boldsymbol{\mathcal{X}}} P_{\boldsymbol{a}}(\boldsymbol{x}, \boldsymbol{y}) [r(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y}) + \gamma \max_{\boldsymbol{b} \in \boldsymbol{A}} q^T(\boldsymbol{y}, \boldsymbol{b})] - q^*(\boldsymbol{x}, \boldsymbol{a}) \end{aligned}$$

前面我们将 Q-Learning 运算表示成一个收缩算子 H，这里可以用它替换公式，得到

$$\begin{aligned} &= (\mathrm{H}q_t)(\boldsymbol{x}, \boldsymbol{a}) - q^*(\boldsymbol{x}, \boldsymbol{a}) \\ &= (\mathrm{H}q_t)(\boldsymbol{x}, \boldsymbol{a}) - (\mathrm{H}q^*)(\boldsymbol{x}, \boldsymbol{a}) \\ &= (\mathrm{H}q_t)(\boldsymbol{x}, \boldsymbol{a}) - (\mathrm{H}q^*)(\boldsymbol{x}, \boldsymbol{a}) \end{aligned}$$

令前面定理的 W 为无穷阶范数 $\|\boldsymbol{x}\|_\infty$，可以得到

$$\|E[F^T(\boldsymbol{x}, \boldsymbol{a}) | F^T]\|_\infty = \|(\mathrm{H}q^T) - (\mathrm{H}q^*)\|_\infty \leqslant \gamma \|q^T - q^*\|_\infty = \gamma \|\Delta^T\|_\infty$$

这样第二个条件就得到了证明。

关于第三个条件，有

$$\begin{aligned} \mathrm{var}[F^T(\boldsymbol{x}) | F^T] &= E[(F^T(\boldsymbol{x}) - E[F^T(\boldsymbol{x})])^2] \\ &= E[\{(r(\boldsymbol{x}, \boldsymbol{a}, X(\boldsymbol{x}, \boldsymbol{a})) + \gamma \max_{\boldsymbol{b} \in \boldsymbol{A}} q^T(\boldsymbol{y}, \boldsymbol{b}) - q^*(\boldsymbol{s}, \boldsymbol{a})) \end{aligned}$$

$$
\begin{aligned}
&-((\mathrm{H}q^T)(\boldsymbol{x},\boldsymbol{a})-q^*(\boldsymbol{x},\boldsymbol{a}))\}^2] \\
&=E[\{r(\boldsymbol{x},\boldsymbol{a},X(\boldsymbol{x},\boldsymbol{a}))+\gamma\max_{\boldsymbol{b}\in\boldsymbol{A}}q^T(\boldsymbol{y},\boldsymbol{b})-(\mathrm{H}q^T)(\boldsymbol{x},\boldsymbol{a})\}^2] \\
&=\mathrm{var}[r(\boldsymbol{x},\boldsymbol{a},X(\boldsymbol{x},\boldsymbol{a}))+\gamma\max_{\boldsymbol{b}\in\boldsymbol{A}}q^T(\boldsymbol{y},\boldsymbol{b})|F^T]
\end{aligned}
$$

我们知道，一般情况下回报值是有界的，且 $\gamma<1$，那么上面公式中两个项目都是有界的，所以对应的方差也必然是有界的，令上式小于某个常数 C_1，我们再找到某个常数 C_2，使得

$$\frac{C_1}{1+\|\Delta^T\|_W^2}\leqslant C_2$$

就可以得到

$$\mathrm{var}[F^T(\boldsymbol{x})|F^T]\leqslant C_1=\frac{C_1}{1+\|\Delta^T\|_W^2}\times(1+\|\Delta^T\|_W^2)\leqslant C_2(1+\|\Delta^T\|_W^2)$$

这样第三个条件也被证明可以满足。通过上面的证明可以得到，Q-Learning 算法可以使 Q 函数依概率 1 得到收敛。通过前面复杂的证明，我们证明出 Q-Learning 的收敛性可以得到保障，当然是有一定条件的。除了 SARSA 法和 Q-Learning 算法，还有其他用于估计价值的方法，例如 Expected-SARSA 方法等，在此不再赘述。

7.8 从表格形式到价值模型

在前面的章节中，我们对价值模型一直有一个很大的限制，那就是它们需要用表格的形式表示。包括 7.7 节的证明，都是基于表格形式才能成立的。表格形式对于求解有很大的帮助，但它也有自己的缺点。如果问题的状态和行动的空间非常大，使用表格表示将难以求解，因为我们需要将所有的状态行动价值都求解出来，才能保证对于任意一个状态和行动，我们都能够得到对应的价值。

以 5.1 节提到的 Atari 游戏为例，经过裁剪处理后，状态空间数为 $256^{84\times84}$ 个，行动数一般不超过 10 个，这样算下来想用表格的形式表达所有的价值，对于现在的计算机系统来说是一个很有挑战的事情。因此，我们需要使用其他的方法。于是研究人员就开始研究一些更合适的表达方式，这时机器学习的方法终于登场，我们可以建立一些模型，并用一个模型来表示状态、行动到值函数的关系。

我们令状态为 $\boldsymbol{s}\in\boldsymbol{S}$，行动为 $\boldsymbol{a}\in\boldsymbol{A}$，最终的值函数为 $v\in\boldsymbol{R}$，那么我们要建立这样一个映射：

$$\boldsymbol{S}\times\boldsymbol{A}\to\boldsymbol{R}$$

这样我们就把强化学习中求解值函数的问题转换成了监督学习的问题，而监督学习是读者熟悉的学习方法，做起来更得心应手。我们可以把这个问题看作回归问题，于是所有可以用于回归问题的模型都可以被我们用上，例如线性回归（Linear Regression）、支持向量回归（Support Vector Regression）、决策树（Decision Tree），以及神经网络（Neural Network）等。在后面的章节中，我们将经常使用一个深层的神经网络完成这个映射的构建。

为了学习值函数模型，我们需要设定模型的目标函数。最简单的目标函数自然是平方损失函数，已知我们通过 Q-Learning 算法得到的状态行动价值为目标价值 v_i，而模型计算得到的价值为估计价值 $v_i'(\boldsymbol{s}, \boldsymbol{a}; \boldsymbol{w})$，那么目标值为

$$\text{obj} = \frac{1}{2}\sum_{i}^{N}(v_i - v_i'(\boldsymbol{s}, \boldsymbol{a}; \boldsymbol{w}))^2$$

其中 $v_i - v_i'(\boldsymbol{s}, \boldsymbol{a}; \boldsymbol{w})$ 也被称为 TD-Error。定义了目标函数，下面就可以利用机器学习中经典的梯度下降法求解：

$$\frac{\partial \text{obj}}{\partial \boldsymbol{w}} = \sum_{i}^{N}(v_i' - v_i)\frac{\partial v_i'}{\partial \boldsymbol{w}}$$

这个公式是不是看上去很眼熟？实际上，如果 $\frac{\partial v_i'}{\partial \boldsymbol{w}} = 1$，那么模型的最优解就等于

$$v_i' = \frac{1}{N}\sum_{i}^{N} v_i$$

也就是所有训练数据的平均值，这和表格版的计算公式一致，也就是说表格版的算法实际上也是一种模型，只不过它的梯度处处为 1。这样，机器学习方法和表格法得到了统一，我们可以认为表格法是一种参数量为 $|\boldsymbol{S}| \times |\boldsymbol{A}|$ 的模型，它的优点是可以完美拟合任意的模型（因为每一点可以独立收敛），缺点是它需要太多的参数。在实践中，表格法有时是无法进行训练的；而对于拟合得到的模型来说，它的参数数量相对可控，我们可以确保模型训练是可行的，但是它的收敛性就不再有保证。

我们知道不同的模型有不同的拟合能力，参数数量少的模型的拟合能力一般比参数数量多的模型差，所以面对不同的真实的值函数，不同的模型可能会拟合出不同的效果，有的结果靠近最优值，有的远离最优值。我们可以用二维平面上的一个点表示值函数的最优点，用一条线表示某个模型能表示的所有值函数的集合，两者的关系如图 7-7 所示。有时模型并不能够拟合得到最优价值，只能够得到近似值。

在 7.7 节中，我们证明了 Q-Learning 方法可以使值函数得到收敛，那么使用模型进行拟合后，这个结论还能不能成立呢？有时结论是成立的，而有时会得到相反的效果。

对于 Q-Learning 算法，我们可以认为每进行一轮更新，得到的值函数会更靠近最优值，而对于基于最小二乘法的优化，我们认为这是一个将值函数投影到模型拟合范围的方法。如图 7-8 所示，经过优化，值函数会投影到模型范围上。

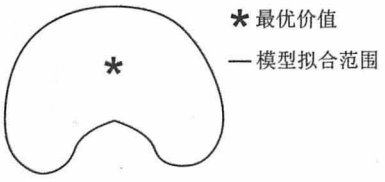


图 7-7 最优值函数和模型拟合范围的关系图

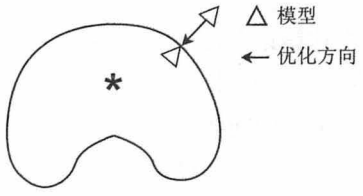


图 7-8 最小二乘法优化后值函数的投影效果

如果只使用 Q-Learning 算法，值函数最终将收敛至最优价值；如果只使用最小二乘法，模型会收敛到最靠近最优价值的位置上。那么看上去，结合这两种方法，我们能够得到最好的效果，但实际上有时并非如此。我们假设当前的参数模型、最优值和拟合范围如图 7-9 所示，首先经过 Q-Learning 的更新，值函数将更新至如图 7-10 所示的位置，模型的效果得到了提升，然而经过最小二乘法的投影后，模型反倒被投影到了更远的位置，如图 7-11 所示。

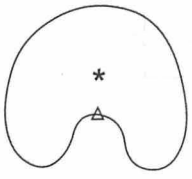


图 7-9 收敛反例的状态图

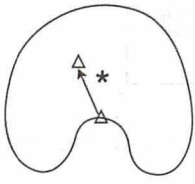


图 7-10 TD 法更新后的状态图

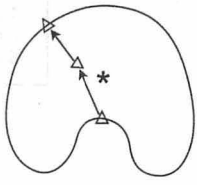


图 7-11 最小二乘法更新后的状态图

所以在一些情况下，这两个步骤并不能保证模型更新的一致收敛，因此模型的能力成了算法收敛的关键。越靠近表格形式的模型，拟合能力越强，也越不容易发生上面所述的情况。深层模型在拟合上的能力很强，这也使它成为了近年来拟合值函数最常见的模型。接下来我们介绍近年来强化学习模型的代表之一：Deep Q Network。

7.9 Deep Q Network

在前面的章节中我们介绍了 Q-Learning 和深层模型的优势，本节就来介绍这个具有极大突破的模型 Deep Q Network，简称 DQN，方法来自论文 *Human-level control through*

deep reinforcement learning[2]。论文主要介绍了如何使用 DQN 网络训练 Agent 在 Atari 游戏平台上尽可能获得更多的分数。基于前面章节的学习，它的整体算法思路就不那么复杂了：在使用 Q-Learning 作为优化算法的前提下，使用深层神经网络表示值函数，直接使用游戏图像和得分进行训练，不再使用其他任何领域的知识。

在这篇论文之前，大多数的强化学习问题都需要研究任务的领域知识，同时需要解决同一序列内样本之间相互关联的问题，以及 Q-Learning 算法中价值估计值与更新值相互关联的问题。这篇论文提出了一系列的解决方案，这些方案虽然并非首创，但是通过将这一系列的设计和改进融入其中，使模型在 Atari 游戏上获得了更好的效果，模型在部分游戏上达到了媲美专家级玩家的效果，整体的算法结构如图 7-12 所示。

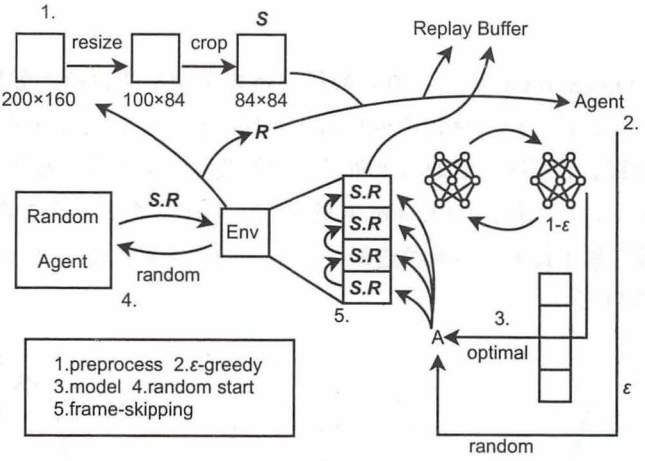


图 7-12　Q-Learning 算法的结构

我们可以从图 7-12 中的序号看出模型的计算流程及作者在模型上的一些小改进。

1. 数据预处理

虽然 DQN 的作者表示算法直接使用了游戏的画面信息进行学习，但实际上为了算法能够更高效地执行，一些预处理工作还是需要的。Atari 游戏的原生尺寸为 210 × 160，每个像素有 128 种颜色，我们首先将其转换成 84 × 84 维度的灰度图。以 Breakout 游戏为例，转换前后的效果如图 7-13 所示，可以看出变换后的图像依然保留了主要的信息，同时减轻了数据处理的负担。

虽然 Atari 游戏是一个动态的游戏，但是每一时刻 Agent 只能从环境中获得 1 帧信息，而这种静态的图像信息很难表示游戏的动态信息。以乒乓球游戏为例，当画面静止时，我们无法判断球要飞向哪一方。为此，算法将收集从当前时刻起的前 N 帧画面，

并将这些信息结合起来作为模型的输入。获得了一定时间内集合的状态信息，模型可以学习到更准确的行动价值，在实验中 N 被设置为 4。

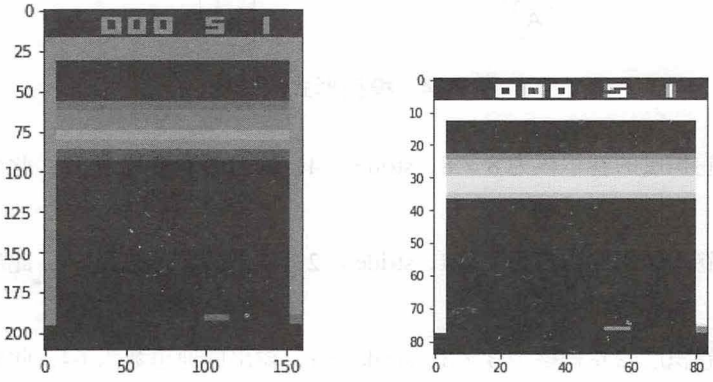


图 7-13　游戏图像预处理效果

除了游戏的画面，游戏的得分也需要做预处理。不同的 Atari 游戏的得分系统不同，有的得分可以上万，有的得分只有几十。为了模型能够更好地拟合长期回报，我们需要将得分压缩到模型擅长处理的范围中，我们将每一轮得到的回报压缩到 $[-1, 1]$。虽然对游戏来说有些不合理，但这样的数值确实更方便模型处理。

2. 环境交互

Atari 游戏的可行状态数量非常多，因此如何更好地探索更多的状态变得十分关键。DQN 采用了 $\epsilon-$greedy 的策略，一开始策略以 100% 的概率随机产生行动，随着训练的不断进行，这个概率将不断衰减，最终衰减至 10%。也就是说，有 90% 的概率执行当前最优策略。这样，从以探索为主的策略逐渐转变成以利用为主的策略，两者得到了很好的结合。

3. 模型结构

前面提到价值模型是从 $|\boldsymbol{S}| \times |\boldsymbol{A}|$ 到 $\boldsymbol{R}$ 的映射，采用这种方法构建模型自然是可以的，但是它还有一些缺点。当模型需要通过值函数求解最优策略时，我们需要计算 $|\boldsymbol{A}|$ 次才能求出，在实际中这样的计算方法效率较低。为了简化计算，我们可以将模型变成 $\boldsymbol{S} \to [\boldsymbol{R}_i]_{i=1}^{|\boldsymbol{A}|}$ 这样的形式，模型的输出为长度为 $|\boldsymbol{A}|$ 的向量，向量中的每一个值表示对应行动的价值估计。这样我们只需要一次计算就可以求出所有行动的价值，无论行动有多少，我们评估价值的时间是一样的。两个模型的表示形式差异如图 7-14 所示。

模型的主体采用了卷积神经网络的结构。

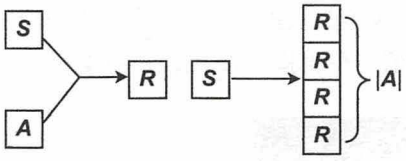


图 7-14　模型形式的转变

第一层卷积层的卷积核为 8×8，stride $= 4$，输出的通道数为 32，此后应用 ReLU 非线性层。

第二层卷积层的卷积核为 4×4，stride $= 2$，输出的通道数为 64，此后应用 ReLU 非线性层。

第三层卷积层的卷积核为 3×3，stride $= 1$，输出的通道数为 64，此后应用 ReLU 非线性层。

第四层是全连接层，输出的维度为 512，此后应用 ReLU 非线性层。

最后一层全连接层得到对应行动的价值估计，输出维度与可行的行动数量相关，一般在 4～18 个。

4. Random Start

有些 Atari 游戏的起始画面是完全一样的，例如 Space Invader，如果每次我们都从一个固定的场景开始做决策，那么 Agent 总要对这些相同的画面进行决策，这显然不利于我们探索更多的画面进行学习。为了增强探索性同时不至于使模型效果变糟，我们可以设定在游戏开始很短的一段时间内，让 Agent 执行随机的行动，这样可以最大程度地获得不同的场景样本。在 5.2.3 节我们分析了 Baselines 中实现的这部分代码，感兴趣的读者可以回顾相关的实现。

5. Frame-Skipping

除了上面对模型的设计，作者还有一个结合现实的考量。虽然我们可以根据游戏的每一个状态给出一个输出，但是实际游戏中玩家并不需要如此高频率的操作。第 1 章提到模拟器可以达到每秒 60 帧的显示速率，但是实际上人类玩家无法实现如此高频率的操作，因此我们可以设定每隔一定帧数执行一次行动，例如每隔 4 帧进行一次行动选择，那么中间的几帧将重复执行前面选择的行动，这样相当于模仿了人类按下某个按钮并持续一段时间的效果。此外，相邻帧之间的画面存在着极大的相似性，对于十分相似的画面，我们通常可以采用相同的行动，因此这样跳过一定帧数的判断也是合理

的。在 5.2.3 节我们分析了 Baselines 中实现的这部分代码，感兴趣的读者可以回顾一下相关的实现。

除了上面介绍的这些要点，还有两个十分重要的创新值得用更多的篇幅介绍，下面就来详细分析这两个创新：Replay Buffer 和 Target Network。

1. Replay Buffer

前面我们已经了解了 Q-Learning 的计算方法。Q-Learning 方法基于当前策略进行交互和改进，更像是一种在线学习（Online Learning）的方法，每一次模型利用交互生成的数据进行学习，学习后的样本被直接丢弃。但如果使用机器学习模型代替表格式模型后再采用这样的在线学习方法，就有可能遇到两个问题，这两个问题也都和机器学习有关。

（1）交互得到的序列存在一定的相关性。交互序列中的状态行动存在着一定的相关性，而对于基于最大似然法的机器学习模型来说，我们有一个很重要的假设：训练样本是独立且来自相同分布的，一旦这个假设不成立，模型的效果就会大打折扣。而上面提到的相关性恰好打破了独立同分布的假设，那么学习得到的值函数模型可能存在很大的波动。

（2）交互数据的使用效率。采用梯度下降法进行模型更新时，模型训练往往需要经过多轮迭代才能收敛。每一次迭代都需要使用一定数量的样本计算梯度，如果每次计算的样本在计算一次梯度后就被丢弃，那么我们就需要花费更多的时间与环境交互并收集样本。

为了解决这两个问题，作者使用了 Replay Buffer 这个数据结构。这个结构被翻译为“样本回放缓存区”，因为这个名字比较拗口，所以在接下来的内容中我们使用英文表达。Replay Buffer 的结构和交互流程如图 7-15 所示。

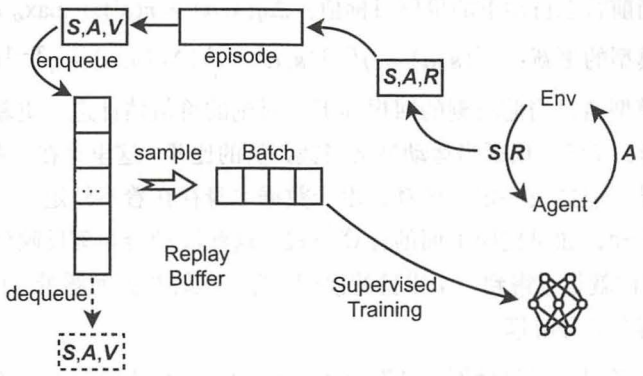


图 7-15 Replay Buffer 的结构图

从图 7-15 中可以看出，Replay Buffer 保存了交互的样本信息，一般来说每一个样本都会保存当前的状态 s、行动 a 和长期累积回报 v。对于其他的算法，Replay Buffer 还会保存其他的信息（第 11 章中将介绍）。Replay Buffer 的大小通常会设置得比较大，例如，将上限设置为可以存储 100 万个样本，这样较长一段时间的样本都可以被保存起来。在训练值函数时，我们就可以从中取出一定数量的样本，根据样本记录的信息进行训练。

总的来说，Replay Buffer 包含了收集样本和采样样本两个过程。收集的样本按照时间先后顺序存入结构中，如果 Replay Buffer 已经存满样本，那么新的样本会将时间上最久远的样本覆盖。而对采样来说，如果每次都取出最新的样本，那么算法就和在线学习相差不多；一般来说，Replay Buffer 会从缓存中均匀地随机采样一批样本进行学习。

均匀采样的好处是什么呢？前面提到我们交互得到的序列在时间维度上存在一定的相关性。我们希望学习得到的值函数能够表示在当前状态行动下的长期收益的期望，然而每一次交互得到的序列，只能代表当前状态-行动下的一次采样轨迹，并不能代表所有可能的轨迹，这样估计的结果就和期望的结果存在一定的差距。随着交互时间不断拉长，这个差距的累积会越来越大。如果完全使用序列的估计值进行训练，某一轮训练时模型会朝着一个序列的估计训练，另一轮训练又会朝着另一个序列的估计训练，那么模型很容易产生较大波动。采用均匀采样后，每次训练的样本通常来自多次交互序列，这样单一序列的波动就被减轻很多，训练效果也就稳定了很多。同时，一份样本也可以被多次训练，提高了样本的利用率。

2. Target Network

模型不稳定的另外一个原因来自算法本身。从 Q-Learning 的计算公式可以看出，算法可以分成如下两个步骤。

（1）计算当前状态行动下的价值目标值：$\Delta q(\boldsymbol{s},\boldsymbol{a}) = r(\boldsymbol{s}') + \max_{\boldsymbol{a}'} q^{T-1}(\boldsymbol{s}',\boldsymbol{a}')$。

（2）网络模型的更新：$q^{T}(\boldsymbol{s},\boldsymbol{a}) = q^{T-1}(\boldsymbol{s},\boldsymbol{a}) + \frac{1}{N}[\Delta q(\boldsymbol{s},\boldsymbol{a}) - q^{T-1}(\boldsymbol{s},\boldsymbol{a})]$。

可以看出模型通过当前时刻的回报和下一时刻的价值估计进行更新，这既像一场自导自演的电影，又像一场既当运动员又当裁判员的比赛。这里存在一些隐患，前面提到数据样本差异可能造成一定的波动，由于数据本身存在着不稳定性，每一轮迭代都可能产生一些波动，如果按照上面的计算公式，这些波动会立刻反映到下一个迭代的计算中，这样我们就很难得到一个平稳的模型。为了减轻相关问题带来的影响，我们需要尽可能地将两个部分解耦。

为此论文作者引入了目标网络（Target Network），它能缓解上面提到的波动性问题。这个方法引入了另一个结构完全一样的模型，这个模型被称为 Target Network，而

原本的模型被称为表现模型（Behavior Network）。两个模型的训练过程如下所示。

（1）在训练开始时，两个模型使用完全相同的参数。

（2）在训练过程中，Behavior Network 负责与环境交互，得到交互样本。

（3）在学习过程中，由 Q-Learning 得到的目标价值由 Target Network 计算得到；然后用它和 Behavior Network 的估计值进行比较得出目标值并更新 Behavior Network。

（4）每当训练完成一定轮数的迭代，Behavior Network 模型的参数就会同步给 Target Network，这样就可以进行下一个阶段的学习了。

通过使用 Target Network，计算目标价值的模型在一段时间内将被固定，这样模型可以减轻模型的波动性。

以上就是 DQN 模型的主要内容，完整的算法流程如下所示。

算法 DQN 算法

Input

初始化容量为 N 的 Replay Buffer：D

初始化状态行动价值模型 Q 和参数 θ

初始化 Target Network $\hat{Q}$ 和参数 θ^-

Start

for episode $= 1, M$ do

初始化环境，得到初始状态 $\boldsymbol{s}_1$，并预处理得到 $\phi_1 = \phi(\boldsymbol{s}_1)$

for $t = 1, T$ do

以 ϵ 的概率随机选择一个行动 $\boldsymbol{a}_t$，或者根据模型选择当前最优

$\boldsymbol{a}_t = \max_{\boldsymbol{a}} Q^*(\phi(\boldsymbol{s}_t), \boldsymbol{a}; \theta)$

执行行动 $\boldsymbol{a}_t$，得到新一轮的状态 $\boldsymbol{s}_{t+1}$ 和回报 $\boldsymbol{r}_{t+1}$

预处理得到 $\phi_{t+1} = \phi(\boldsymbol{s}_{t+1})$

将 $\{\phi_t, \boldsymbol{a}_t, \boldsymbol{r}_{t+1}, \phi_{t+1}\}$ 存储到 D 中

从 D 中采样一批样本 $(\phi_j, \boldsymbol{a}_j, \boldsymbol{r}_{j+1}, \phi_{j+1})$

计算 $\boldsymbol{y}_j = \begin{cases} \boldsymbol{r}_{j+1} & \text{for terminal } \phi_{j+1} \\ \boldsymbol{r}_{j+1} + \gamma \max_{\boldsymbol{a}'} Q(\phi_{j+1}, \boldsymbol{a}'; \theta^-) & \text{for non-terminal } \phi_{j+1} \end{cases}$

根据目标函数 $(\boldsymbol{y}_j - Q(\phi_j, \boldsymbol{a}_j; \theta))^2$ 进行梯度下降法求解

每隔 C 轮完成参数更新 $\theta^- \leftarrow \theta$

end for

end for

从论文中的介绍和实际的效果看，DQN 已经实现了很大的飞跃，效果有了显著的提高，但这还不够，第 8 章我们将继续介绍 DQN 的改进算法。

7.10 总结

本章我们学习了无模型的学习算法，以及其中的经典算法 DQN，下面让我们回顾一下。

（1）在无模型问题中，我们将无法使用状态转移概率进行策略求解。

（2）蒙特卡罗法通过统计完整序列的长期回报来完成策略评估的过程。

（3）TD 法通过当前时刻的回报和下一时刻开始的长期回报估计来完成策略评估的过程。

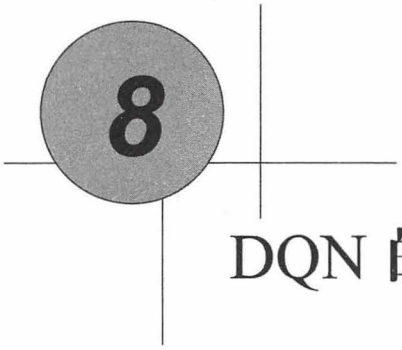
（4）SARSA 法和 Q-Learning 法分别代表了 On-Policy 和 Off-Policy 的 TD 法。

（5）DQN 结合了 Q-Learning 的价值估计方法和深层模型较强的拟合效果，同时还结合了 Replay Buffer 和 Target Network 这样的结构，在 Atari 游戏上获得了不错的结果。

7.11 参考资料

[1] Mnih V, Kavukcuoglu K, Silver D, et al. Playing Atari with Deep Reinforcement Learning[J]. Computer Science, 2013.

[2] Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. Nature, 2015, 518(7540):529.



DQN 的改进算法

第 7 章介绍了 Q-Learning 和 DQN 的算法细节。本章继续介绍与 DQN 相关的算法，这些算法都是基于 DQN 算法的扩展算法，它们从不同的角度解决了 DQN 中存在的问题，也获得了更好的效果。

为了减少模型的波动性，Double Q-Learning 可以使用不同的模型完成最优行动的选择和价值估计两个工作，使价值估计的波动降低；第 7 章介绍了 Replay Buffer 的概念和作用，本章将介绍基于它的改进算法 Priority Replay Buffer，它可以更高效地利用存储的样本；DQN 的模型输出的是状态行动值函数 $q(\boldsymbol{s}, \boldsymbol{a})$，它用一个单一的数值表示了价值期望，Duel DQN 将 $q(\boldsymbol{s}, \boldsymbol{a})$ 拆解成 $v(\boldsymbol{s}) + \text{adv}(\boldsymbol{s}, \boldsymbol{a})$ 两部分；而 Distributional DQN 则以类似直方图的形式表示了价值的分布；Noisy DQN 则通过增加参数的随机性增强模型的探索性能；Rainbow 模型则将前面的改进融合在了一起。

8.1 Double Q-Learning

本节介绍 Double Q-Learning，算法来自论文 *Deep Reinforcement Learning with Double Q-learning*[1]。在第 7 章介绍 DQN 时，曾提到模型中的一个核心结构 Target Network。DQN 使用了两个结构相同的网络：Behavior Network 和 Target Network。通过在一段时间内固定 Target Network 的参数，Q-Learning 方法的目标价值能够得到一定的固定，这样模型也能够获得一定的稳定性。

虽然这个方法提升了模型的稳定性，但它并没有解决另外一个问题：Q-Learning 对价值过高估计的问题。我们知道 Q-Learning 在计算时利用了下一时刻的最优价值，所

以它通常在计算时给出了一个状态行动的估计上限。由于训练过程中模型并不够稳定，因此对上限的估计可能会存在一定的偏差。如果偏差是一致的，也就是说，每个行动都拥有相近的偏差，那么偏差对模型效果的影响相对较小；如果偏差是不一致的，那么这个偏差会造成模型对行动优劣的判断偏差，这样就会影响模型的效果。

我们已经知道 Target Network 求解价值目标值的公式

$$\boldsymbol{y}_j = \boldsymbol{r}_{j+1} + \gamma \max_{\boldsymbol{a}'} Q(\boldsymbol{s}_{j+1}, \boldsymbol{a}'; \theta^-)$$

其中 θ^- 表示 Target Network 的参数。将公式进一步展开，可以得到更详细的公式内容：

$$\boldsymbol{y}_j = \boldsymbol{r}_{j+1} + \gamma Q(\boldsymbol{s}_{j+1}, \operatorname{argmax}_{\boldsymbol{a}'} Q(\boldsymbol{s}_{j+1}, \boldsymbol{a}'; \theta^-); \theta^-)$$

公式展开后，我们发现采用 Target Network 后，模型在选择最优行动和计算目标值时依然使用了同样的参数模型，这样必然会造成对价值的过高估计。为了尽可能地减少过高估计的影响，一个简单的办法就是把选择最优行动和估计最优行动两部分的工作分离，我们用 Behavior Network 完成最优行动的选择工作，这样就可以得到

$$\boldsymbol{y}_j = \boldsymbol{r}_{j+1} + \gamma Q(\boldsymbol{s}_{j+1}, \operatorname{argmax}_{\boldsymbol{a}'} Q(\boldsymbol{s}_{j+1}, \boldsymbol{a}'; \theta); \theta^-)$$

通过这样的变化，算法在三个环节的模型安排如下。

（1）采样：Behavior Network$Q(\theta)$。

（2）选择最优行动：Behavior Network$Q(\theta)$。

（3）计算目标价值：Target Network$Q(\theta^-)$。

经过这样的变换，模型在过高估计的问题上得到了缓解，稳定性也就得到了提高。

8.2 Priority Replay Buffer

Priority Replay Buffer 是一种针对 Replay Buffer 的改进结构，它来自论文 *Prioritized Experience Replay*[2]。在 7.9 节我们了解了 Replay Buffer，它能够在提高样本利用率的同时减少样本的相关性。那么它还存在什么问题吗？在采样样本时，每一个样本都会以相同的概率被采样出来。也就是说，每一个样本，都会以相同的频率被学习。但实际上，每一个样本的难度是不同的，学习每一个样本得到的收获也不同。有的样本相对简单，表现相对较好，学习得到的收获不会太高；有的样本则有些困难，表现也相对较

差，经过学习后模型会有显著的提高。如果平等地看待每一个样本，就会在那些简单的样本上花费比较多的时间，而学习的潜力没有被充分挖掘出来。

Priority Replay Buffer 则很好地解决了这个问题。它会根据模型对当前样本的表现情况，给样本一定的权重，在采样时样本被采样的概率就和这个权重有关。交互时表现得越差，对应的权重越高，采样的概率也就越高；反过来，交互时表现得越好，那么权重也就越低，采样的概率也就越低。

这样，那些使模型表现不好的样本可以有更高的概率被重新学习，模型就会把更多的精力放在这些样本上，模型的学习效率就会有一定的提升。现在，Priority Replay Buffer 已经被广泛应用在 Q-Learning 的算法中，而且这个思想也被神经科学界证实，在人脑上也存在类似的学习机制。

接下来，我们来看看它在代码中的具体实现方法。Baselines 项目的 baselines/deepq/replay_buffer.py 中实现了 Replay Buffer 和 Priority Replay Buffer 两个结构。从算法原理上看，两者的差别在以下两个方面。

（1）为每一个存入 Replay Buffer 的样本设定一个权重。

（2）使用这个权重完成采样过程。

我们先来解决第 2 个问题。假设权重已经得到，我们就可以采用轮盘赌的方法进行样本的采样。如果像 Replay Buffer 一样直接使用数组保存每一个样本和权重，它的更改样本和采样的流程如图 8-1 所示。

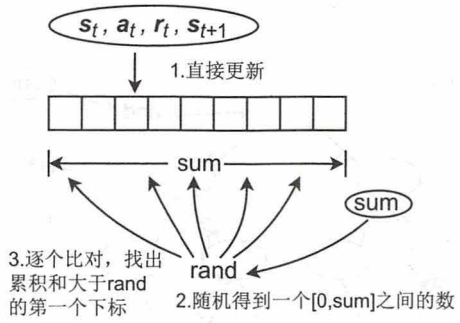


图 8-1　利用数组实现 Priority Replay Buffer 功能的流程

流程分为 3 步：在添加样本到 Replay Buffer 时，直接将其写入而不做任何处理，同时更新 Replay Buffer 中所有样本的权重总和 sum；在采样时随机得到一个 [0,sum] 之间的数，那么从 Replay Buffer 算起，累积权重和超过随机值的第一个样本就被选择为待训练的样本，它的运算复杂度如下。

- 更改复杂度：$O(1)$，只需要在指定的位置写入信息。

- 采样复杂度：$O(N)$，需要按序扫描指定的位置才能结束，最坏的情况下可能会扫描整个数组。

从上面的分析可以看出，采样的复杂度有些高，Replay Buffer 的大小一般比较大，第 7 章曾提到很多任务都会将其设置为 100 万，那么扫描整个数组的代价就会比较大。为了能快速实现样本集合中一部分样本的权重修改和集合的采样，我们可以采用线段树这个数据结构实现这个功能。线段树能够以比较快的速度更新局部的统计信息，同时可以利用这些信息快速查找满足某些特征的数据。线段树的结构如图 8-2 所示。

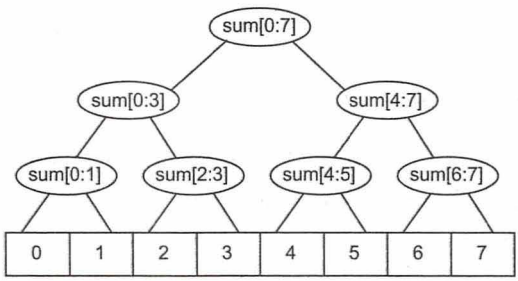


图 8-2　线段树示意图

线段树的主体是由一棵树构成的。它的叶子节点存放着每一个元素个体的信息，例如它的权重，而线段树的非叶子节点则存放着其子节点的权重汇总。如果这是一颗求和的线段树，那么非叶子节点存放着其叶子节点的权重总和；如果这是一颗求最小值的线段树，那么非叶子节点存放着其叶子节点的最小值。使用线段树后的更新与采样的流程如图 8-3 所示。

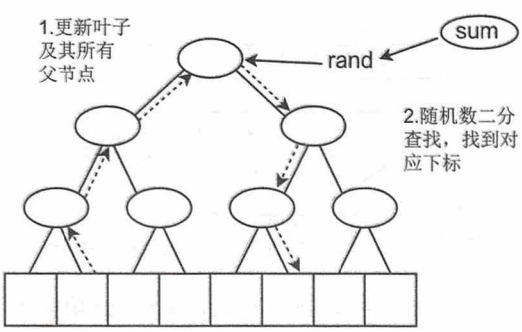


图 8-3　利用线段树实现 Priority Replay Buffer 功能的流程

它的流程同样有 3 步。在添加样本到 Replay Buffer 时，不光记录样本信息，还要沿着样本叶子出发，依次找到叶子节点的父节点，更新父节点的权值；在采样时同样得到一个随机数，然后利用树的结构二分查找就可以快速定位被选到的样本，它的运算复杂度如下。

- 更改复杂度：$O(\log N)$，需要更新叶子节点的所有父节点的信息。
- 采样复杂度：$O(\log N)$，只需要从根出发访问指定的叶子。

对比发现，线段树在采样上明显快于数组，而更新的代价又没有增加太多，因此它可以替代数组使用。这样第二个问题就得到了解决。

让我们回到第一个问题。Priority Replay Buffer 的主要思路是增加有价值样本出现的频率。那么我们该如何定义“有价值”这个概念呢？最直观的想法是，能让模型学习到更多的内容，这个样本一定更有价值。那么，什么样的样本能让模型学习到更多呢？当然是能产生更大的 TD-Error 的，即估计值与目标值的差距越大，样本越有价值。所以我们可以将 TD-Error 作为样本重要性的衡量指标。TD-Error 越高，样本出现的概率就越高。

虽然 TD-Error 是一个很合理的指标，但是由于我们的模型处在学习的过程中，在样本放入 Priority Replay Buffer 时样本的 TD-Error 和它被取出时的 TD-Error 有可能不一样。所以可能会出现一种情况：一个样本在放入时 TD-Error 很高，但是随着模型的不断学习，模型对这个样本的实际 TD-Error 已经降低，但它还是会以很高的概率出现。一个最合理的方法，是在我们从 Priority Replay Buffer 取出样本时，将它的 TD-Error 重新计算一遍，但是这样做的代价实在太大，所以我们只好使用样本放入 Priority Replay Buffer 时的 TD-Error 作为近似。

既然我们已经知道这个指标并不完全可靠，就需要其他的方法辅助。因此我们要确保 TD-Error 较高的样本出现的概率更高，同时也要使那些 TD-Error 较低的样本以一定的概率出现。论文中定义的一个计算样本出现概率的方法为

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

其中 p_i 表示每个样本在计算时的 TD-Error。α 可以调整样本 TD-Error 的重要性。当 $\alpha = 1$ 时，相当于我们直接使用 TD-Error 数值；当 $\alpha < 1$ 时，我们就可以削弱高 TD-Error 样本的影响，增强低 TD-Error 样本的影响。这样我们就可以通过这个参数调整 Priority Replay Buffer 的表现。

除了上面提到的两个不同，实际上 Priority Replay Buffer 还做了第三个改变，它提供了一个参数，用于调整每个样本对模型更新的影响。对 Replay Buffer 来说，每一个样本是被等概率取出的，它们对模型的更新也是等权重的；而 Priority Replay Buffer 的样本是非等概率取出的，它的样本服从另一个难以清楚地描述的分布，所以我们对模型的更新是有偏差的。为了使我们的更新无偏，可以采用重要性采样的方法使更新变

得无偏，对应的公式为

$$\begin{aligned} E_{i\sim \text{RB}}[\nabla J] &= E_{i\sim \text{PRB}}[\frac{P_{\text{RB}}(i)}{P_{\text{PRB}}(i)}\nabla J] \\ &= E_{i\sim \text{PRB}}[\frac{\frac{1}{N}}{P_{\text{PRB}}(i)}\nabla J] \\ &= E_{i\sim \text{PRB}}[\frac{1}{N\cdot P_{\text{PRB}}(i)}\nabla J] \end{aligned}$$

其中 N 表示 Replay Buffer 存放的样本数量。所以我们可以在每一个被学习的样本前增加一个权重 $\frac{1}{N\cdot P_{\text{PRB}}(i)}$，这样就可以使更新变得无偏。

总感觉哪里有点不对劲……我们使用 Priority Replay Buffer 的目的不正是让更新变得有偏吗？为什么还要削弱它的优势呢？实际上，这个纠正是为了说明我们可以通过一些设定让它变回 Replay Buffer 那样的更新方式，这样虽然没有带来任何好处，但也没有任何坏处。那么我们就可以根据实际问题调整这个权重，让它在两种更新效果之间做一个权衡，既能确保提升样本的利用率，又能确保结果不会带来太大的偏差，新的权重更新公式变为

$$w_i = (\frac{1}{N\cdot P_{\text{PRB}}(i)})^{\beta}$$

这样当 $\beta = 1$ 时，更新效果实际上等同于 Replay Buffer；当 $\beta < 1$ 时，Priority Replay Buffer 就能够发挥作用了。

从另一个角度讲，很多理论推导都是基于 Replay Buffer 的，我们所做的改进主要希望训练速度能够增加，至于方法的收敛性却并不确定。所以我们还是希望 Priority Replay Buffer 能够最终变成 Replay Buffer，所以可以让 β 在训练开始时赋值为一个小于 1 的数，然后随着训练迭代数的增加，让 β 数不断变大，并最终到达 1。这样我们既可以确保训练速度能够增加，又可以让模型的收敛性得到保证。

以上是 Priority Replay Buffer 的算法原理，在此我们可以做一个总结。

（1）在样本存入 Replay Buffer 时，计算 $P(i) = \frac{p(i)^{\alpha}}{\sum_j p(j)^{\alpha}}$。

（2）在样本取出时，以第 1 步计算得到的概率进行采样。

（3）在更新时，为每一个样本添加 $w_i = (\frac{1}{N\cdot P_{\text{PRB}}(i)})^{\beta}$ 的权重。

（4）随着训练的进行，让 β 从某个小于 1 的值渐进地靠近 1。

从算法的思想到最终实现，我们经历了许多波折，可以看出很多算法在设计时需要同时兼顾“求新”和“求稳”两个目标，这样才能在更大程度上提高算法的效果。如果去看代码实现，读者一定会发现在实现中 Priority Replay Buffer 还包含其他未提到的结构（例如最小值线段树），这些内容欢迎读者进行探索。

8.3 Dueling DQN

Dueling DQN 是一种基于 DQN 的改进算法，它的主要突破点在于利用模型结构将值函数表示成更细致的形式，这使得模型能够拥有更好的表现。它来自论文 *Dueling network architectures for deep reinforcement learning*[3]。接下来，我们就来看看它的改进形式。

经过前面的介绍，我们可以给出这个公式并定义一个新的变量：

$$q(\boldsymbol{s}_t, \boldsymbol{a}_t) = v(\boldsymbol{s}_t) + A(\boldsymbol{s}_t, \boldsymbol{a}_t)$$

也就是说，基于状态和行动的值函数 q（以下简称为 q）可以分解成基于状态的值函数 v（以下简称为 v）和优势函数（Advantage Function）A（以下简称为 A）。由于存在

$$E_{\boldsymbol{a}_t}[q(\boldsymbol{s}_t, \boldsymbol{a}_t)] = v(\boldsymbol{s}_t)$$

所以如果所有状态行动的值函数不相同，一些状态行动的价值 $q(\boldsymbol{s}, \boldsymbol{a})$ 必然会高于状态的价值 $v(\boldsymbol{s})$，当然也会有一些状态行动对低于价值，于是优势函数可以表示出当前行动和平均表现之间的区别：如果优于平均表现，那么优势函数为正；反之则为负。

既然概念上有这样天然的分解，那么在设计模型时能不能也采用这样的结构呢？当然可以。于是我们在保持网络主体结构不变的基础上，将原本网络中的单一输出变成两路输出，一个输出用于输出 v，它是一个一维的标量；另一个输出用于输出 A，它的维度和行动数量相同。最后将两部分的输出加起来，就是原本的 q 值。改变输出结构后，我们只需要对模型做很少的改变即可实现功能：模型前面的部分可以保持不变，模型后面的部分从一路输出改变为两路输出，最后再合并成一个结果。模型结构的改变如图 8-4 所示。

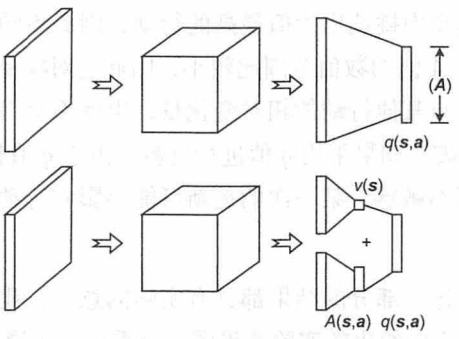


图 8-4 Dueling DQN 结构变化图

难道我们单纯地将它们分解就可以了吗？当然不行。我们还需要对这两部分的输出做一定的限制。如果我们不对这两个输出做限制，那么当 q 值一定时，v 和 a 有无穷种可行组合，而实际上只有很小一部分的组合是合乎情理、接近真实数值的。为了解决这个问题，我们可以对 A 函数做限定。我们知道 A 函数的期望值为 0：

$$\begin{aligned} E_{\boldsymbol{a}}[A(\boldsymbol{s}_t, \boldsymbol{a}_t)] &= E_{\boldsymbol{a}}[q(\boldsymbol{s}_t, \boldsymbol{a}_t) - v(\boldsymbol{s}_t)] \\ &= v(\boldsymbol{s}_t) - v(\boldsymbol{s}_t) \\ &= 0 \end{aligned}$$

就可以对输出的 A 值进行约束，例如将公式变成

$$q(\boldsymbol{s}_t, \boldsymbol{a}_t) = v(\boldsymbol{s}_t) + (A(\boldsymbol{s}_t, \boldsymbol{a}_t) - \frac{1}{|A|}\sum_{\boldsymbol{a}'} A(\boldsymbol{s}_t, \boldsymbol{a}'_t))$$

让每一个 A 值减去当前状态下所有 A 值的平均数，就可以保证前面提到的期望值为 0 的约束，从而增加了 v 和 A 的输出稳定性。

进行这样的输出分解有什么好处呢？首先，通过这样的分解，我们不但可以得到给定状态和行动的 q 值，还可以同时得到 v 值和 A 值。这样如果在某些场景需要使用 V 值时，我们同样可以获得 v 值而不用再训练一个网络。同时，通过显式地给出 v 函数的输出值，每一次更新时，我们都会显式地更新 v 函数。这样 V 函数的更新频率就会得到确定性的增加。对于单一输出的 q 网络来说，它的更新就显得有些晦涩。

其次，从网络训练的角度来看，我们从原本需要训练 $|A|$ 个取值为 $[0, \infty]$ 的数值，变成了训练一个取值为 $[0, \infty]$ 的数值，和 $|A|$ 个均值为 0，实际取值为 $[-C, C]$ 的数值，对网络训练来说，后者显然是更友好且容易的。同时，对一些强化学习的问题来说，A 值的取值范围远比 v 值小，这样将两者分开训练更容易保持行动之间的排列顺序。由于我们需要从所有的行动中挑选出价值最高的行动，因此不同行动之间的价值需要保持一定的区分度。由于 A 值的数值范围比较小，因此它对模型更新更敏感，这样模型在更新时会更容易考量与其他行动的相对变化量，也就不会因为某一次的更新使得原本的行动排序被意外打破。如果采用 q 值进行更新，由于 q 值相对于 A 值可能会很大，因此 q 值对微小的更新不敏感，某一次的更新可能会影响行动的原本排序，从而对策略造成一定的波动。

将值函数分解后，每一部分的结果都具有实际的意义，我们也可以从中挖掘出很多有价值的信息。从论文中给出的实验效果图可以看出，将模型分为 A 函数和 v 函数后，两个函数会展示出不同的特点。通过反卷积操作得到两个函数值对原始图像输入

的梯度后，可以发现 v 函数对游戏中的所有关键信息都十分敏感，而 A 函数只对和行动相关的信息敏感，如图 8-5 所示。

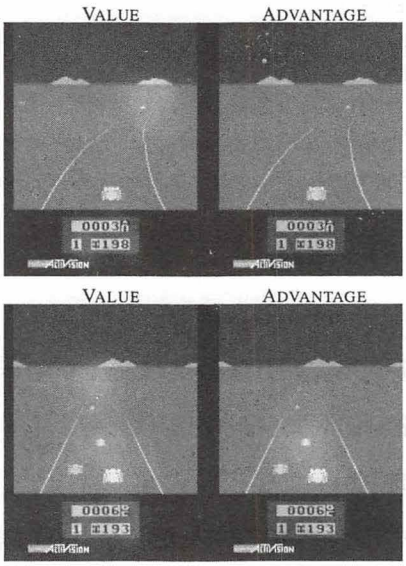


图 8-5 Dueling Network 的反卷积敏感区域图

图 8-5 的上面两幅图中，玩家附近没有障碍物时，价值敏感区域比较单一，由于行动之间对结果的影响差距不大，所以优势函数中没有太敏感的区域；而图 8-5 的下面两幅图中，玩家附近有一些障碍物，因此优势函数的敏感区域变得多了起来。两个输出敏感图的差异说明了价值和优势之间的差别。

8.4 解决 DQN 的冷启动问题

本节要解决 DQN 网络冷启动的问题。一直以来，强化学习问题都遇到了“冷启动”的问题。对于以值函数为核心的 Q-Learning 算法来说，前期的算法迭代很难让模型快速进入一个相对理想的环境，更何况由于前期值函数估计存在较大偏差，与环境交互得到的采样与最优策略存在一定的差别，这更增加了学习的难度。那么有没有能提高训练初期的学习效率的办法呢？论文 *Deep Q-learning from Demonstrations*[4] 为我们探索了强化学习和监督学习结合的一种方案。

作者的主要思路是利用预先准备好的优质采样轨迹加快模型前期的训练速度。这个思路是非常直观的。对一个未经任何训练的模型来说，模仿优质采样轨迹的行动是一个不错的选择，因为模型训练初期 Agent 的策略相对较差，从交互序列中学习的效

果不会很好，而优质行动本身来自较强的策略，这样就相当于直接站在巨人的肩膀上，学习速度自然会快很多。只要我们有办法取得一定数量的优质轨迹，就可以通过监督学习完成与环境交互前的预训练。

除了使用监督学习预训练，我们还要使用强化学习完成 DQN 模型原本方法中的训练。最终模型的目标函数变成了多个学习目标结合的形式：

$$J(q) = J_{\mathrm{DQ}}(q) + \lambda_1 J_n(q) + \lambda_2 J_E(q) + \lambda_3 J_{L2}(q)$$

这个目标函数包含了 4 个子目标。

（1）$J_{\mathrm{DQ}}(q)$：Deep Q-Learning 的目标函数。

（2）$J_n(q)$：以 n 步回报估计法为目标的 Q-Learning 目标函数。

（3）$J_E(q)$：利用准备数据进行监督学习的目标函数。

（4）$J_{L2}(q)$：L2 正则的目标函数。

其中，λ_n 用于平衡不同目标函数之间的权重。相信读者对第 1 和第 4 个子目标比较熟悉，下面就来重点介绍第 2 和第 3 个子目标。首先介绍 $J_n(q)$。Q-Learning 在计算对某个状态-行动的值函数的估计值时，采用了如下估计法：

$$q'(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{r}_{t+1} + \max_{\boldsymbol{a}} q(\boldsymbol{s}_{t+1}, \boldsymbol{a})$$

前面曾介绍过，对于序列比较长的问题，这种方法有一个缺陷，那就是价值需要一定时间的训练才能跳过前面波动较大的时期，进入相对稳定的更新时期。为了使模型的更新进行得更快，我们可以将上面的公式进一步展开，不仅使用下一时刻的回报，而是将此后更多时刻的回报加入目标值中。公式如下所示：

$$q'(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{r}_{t+1} + \gamma \boldsymbol{r}_{t+2} + \cdots + \gamma^{n-1} \boldsymbol{r}_{t+n} + \max_{\boldsymbol{a}} \gamma^n q(\boldsymbol{s}_{t+n+1}, \boldsymbol{a})$$

这个估计方式被称为 n 步回报的估计法，可以看出这个方法结合了蒙特卡罗法的特点，在公式中使用了更多真实的交互回报，这样更容易在模型训练早期将值函数做快速更新，一般来说，n 可以设置为 10。这种更新方式还会在后面的章节中再次出现。

其次介绍 $J_E(q)$。一般来说，事先准备好的数据是比较有限的，很难支撑一个完整模型的训练，因此它必然只能影响很小一部分的状态行动值。如果它不能尽可能地覆盖更多的状态，那么这些数据反而有可能对模型造成不好的影响。同时，这些准备好的数据中也可能存在一些噪声，其中的行动并不是真正的行动。为了避免上面提到的问

书籍是人类进步的阶梯

题，监督学习的目标函数被定义为如下形式：

$$J_E(Q) = \max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E)$$

其中 a_E 表示当前状态下专家给出的行动，$l(x,y)$ 函数是一个指示函数，当模型选择的行动与 a_E 相同时，函数值为 0；当选择不同时，函数值为某个固定值。当 $a = a_E$ 时，目标函数等于 0，表示模型的决策与准备数据相同；如果 $a \neq a_E$，则说明其他某个行动的价值至少不弱于专家行动太多，这对模型来说是一个比较合理的约束。

这样我们就了解了目标函数 4 个部分的含义。4 个子目标涵盖了监督学习和强化学习各方面的目标。除此之外，算法还有一些其他的设定：

- 由于为预训练准备的样本质量比较高，我们可以对其进行反复利用，因此它们在训练过程中不会被换出来，全部永久存在于 Replay Buffer 中。
- 同样，由于准备数据和交互数据的来源和质量不同，因此在从 Replay Buffer 中随机抽取样本时，准备数据和交互得到的数据拥有不同的采样概率；模型一开始只接受准备数据的学习，不进行模型模拟采样，这样确保了模型前期的快速成长。
- 算法也使用了常见的 Target Network、Priority Replay Buffer 等技巧。

模型结果显示，该方法在早期训练上明显超过其他方法，同时在最终结果上也超过了很多当前表现最优的模型，获得了很不错的成绩。完整的算法流程如下所示：

算法 Deep Q-learning from Demonstrations

Input

D_{replay}：专家行动数据作为初始化的 Replay Buffer

θ：Behaviour Network 的参数

θ'：Target Network 的参数

τ：更新 Target Network 的频率

k：预训练的轮数

Start

For steps $t \in \{1, 2, \ldots, k\}$ do

　从 D_{replay} 中进行优先采样，得到一个 mini-batch 的数据

　使用 Target Network 计算损失函数 $J(Q)$

　对网络 θ 进行梯度下降更新参数

　当 t **mod** $\tau = 0$ 时，进行 Target Network 的更新：$\theta' \leftarrow \theta$

for steps $t \in \{1, 2, \ldots\}$ do

　利用 Behaviour Networkθ 进行交互，获得执行数据 (s, a, r, s')

　将数据存入 D_{replay}，同时根据容量情况替换老的数据

書籍是人类进步的阶梯

从 D_{replay} 中进行优先采样，得到一个 mini-batch 的数据
使用 Target Network 计算损失函数 $J(Q)$
对网络 θ 进行梯度下降更新参数
当 $t \bmod \tau = 0$ 时，进行 Target Network 的更新：$\theta' \leftarrow \theta$

8.5 Distributional DQN

本节介绍 Distributional DQN，这个算法同 Dueling DQN 类似，都是要对价值模型的结构进行改进。它来自论文 *A Distributional Perspective on Reinforcement Learning*[5]。在此之前，我们介绍了 DQN 的基本思想，那就是通过神经网络完成值函数的拟合。状态行动值函数本质上计算了基于某个特定策略对未来的长期回报的期望，这个期望值的估计既不容易，也不细致。Distributional DQN 模型希望通过建立更复杂细致的值函数，让估计结果更细致可信。

8.5.1 输出价值分布

让我们先回忆 Bellman 公式的形式：

$$\begin{aligned} q(\boldsymbol{s},\boldsymbol{a}) &= r(\boldsymbol{s},\boldsymbol{a}) + \sum_{\boldsymbol{s}'} p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \sum_{\boldsymbol{a}'} \pi(\boldsymbol{a}'|\boldsymbol{s}) \gamma q(\boldsymbol{s}',\boldsymbol{a}') \\ &= r(\boldsymbol{s},\boldsymbol{a}) + \sum_{\boldsymbol{s}'} p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \gamma E_{\boldsymbol{a}' \sim \pi(\boldsymbol{a}'|\boldsymbol{s})}[q(\boldsymbol{s}',\boldsymbol{a}')] \\ &= r(\boldsymbol{s},\boldsymbol{a}) + \gamma E_{\boldsymbol{s}',\boldsymbol{a}' \sim p,\pi}[q(\boldsymbol{s}',\boldsymbol{a}')] \end{aligned}$$

由此可见，目前模型的估计结果是基于策略和状态转移两个概率分布的价值期望，由于综合了太多的信息，这个数值就显得有些不够具体，如果想要得到更详细的信息，那么模型就不能仅输出一个期望值，而需要将价值的分布估计出来。这样模型的结果就会做出如图 8-6 所示的变化。

如果我们能将价值的分布估计出来，这个结果将变得更有意义。但是分布中拥有太多的信息，精确地表示分布并不是一件容易的事。如果我们把价值分布想象成一个比较常见的分布，那么原本的价值模型同样可以表示为分布的形式。例如，假设价值分布是一个方差固定的高斯分布，那么我们只需要估计出分布的均值，就可以知道分布的形式。但是这样的假设可能和真实情况存在一定差距，所以我们要使用更弱的假设求出分布。

书籍是人类进步的阶梯

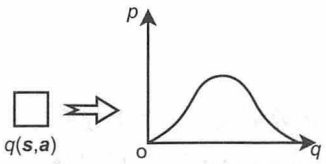


图 8-6 分布估计变化

为了确保分布不受太多的限制，同时又可以减少分布的计算量，我们选择一种简单直观的方法：直方图。假设绝大多数的价值最终值落在区间 $[V_{\min}, V_{\max}]$ 之内，同时限定区间内每一段的价值范围相等，那么我们只需让模型输出给定值函数区间的概率，也就是 $P(V)$，就可以表示出直方图分布的形式。这样我们的价值分布估计又经历了如图 8-7 所示的转变。

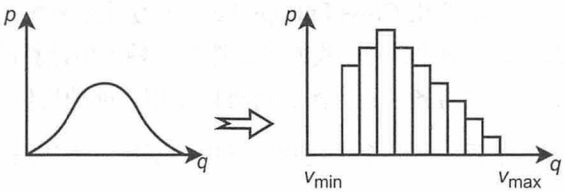


图 8-7 用直方图表示分布

采用直方图的方法有一个好处，那就是它可以近似很多不太常规的分布形式，分布的表示范围比起前面提到的正态分布要大很多。前面提到直方图的最大值和最小值是限定的，其中的区间也是等距表示的。假设直方图一共有 N 段，那么每一段的间隔为

$$\Delta \boldsymbol{z} = \frac{V_{\text{MAX}} - V_{\text{MIN}}}{N-1}$$

于是直方图中表示的数值采样集合为 $\{\boldsymbol{z}_i = V_{\text{MIN}} + i\Delta \boldsymbol{z} : 0 \leqslant i < N\}$，这个集合共有 N 个元素。我们的模型要输出 N 个值的向量，每一个值表示这其中一个价值采样点出现的概率，于是模型可以用下面的映射表示

$$\theta : \boldsymbol{X} \times \boldsymbol{A} \to \boldsymbol{R}^N$$

不同于 DQN 输出实数，直方图模型会通过 Softmax 层输出每一个价值采样点的概率，这样可以得到价值为 $\boldsymbol{z}_i$ 时，对应的概率为 $p_i(x,a) = \frac{e^{\theta_i(x,a)}}{\sum_j e^{\theta_j(x,a)}}$，模型的输出结构也就变得十分清晰。如果目标价值分布和估计价值分布的价值范围和价值采样点完全相同，那么两者的比较就相当于同一个概率分布结构下不同概率形式的比较，也就可以使用 Cross Entropy Loss 作为价值模型的目标函数。

书籍是人类进步的阶梯

8.5.2 分布的更新

解决了输出的表示问题，还要解决另一个十分重要的问题，那就是更新。在前面的 DQN 模型中，我们可以使用 Q-Learning 公式计算目标价值，得到的结果为目标期望价值；而在 Distributional DQN 模型中，我们的计算过程则变得有些复杂。

（1）计算 $t+1$ 时刻每一个行动的价值分布。

（2）将第 1 步的价值期望和 $t+1$ 时刻的回报相加，并选出期望最大的行动。

（3）将期望最大的行动的价值分布表示出来。

我们发现第 3 步计算后，得到的分布可能和原始的分布形式不同。不光是价值范围不同，价值采样点也会发生偏移，这样目标价值分布和估计分布将很难比较。如果允许这样的不同，我们就不能使用 Cross Entropy Loss 作为目标函数，选择其他的函数将会使模型变得更复杂，为了简化计算，我们可以选择一种近似计算的方法，那就是将价值范围和采样点对齐。只要对齐了直方图，就可以使用之前的目标函数进行计算。

直方图分布对齐的过程涉及几个子过程，其中的变换过程如图 8-8 所示。

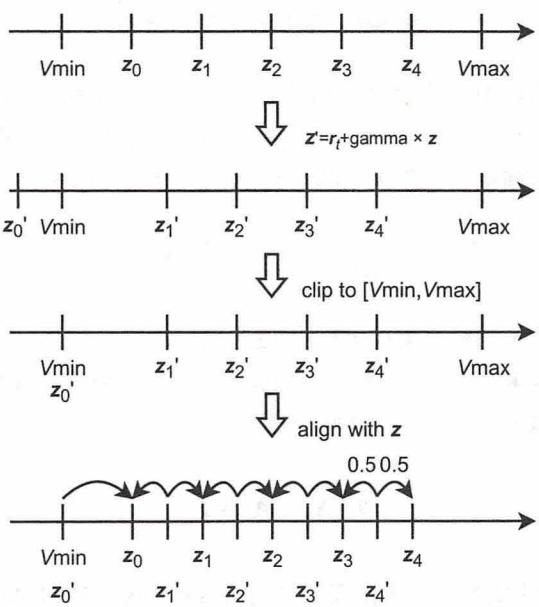


图 8-8 价值分布更新过程

在对齐前，我们要先计算出最优的行动。由于价值的范围和采样点已经确定，那么对于不同的行动，可以计算出每一个价值采样点 z_i 的概率 $p(z_i)$，这样也可以求出当前

书籍是人类进步的阶梯

行动 $\boldsymbol{a}$ 的价值期望：

$$q(\boldsymbol{s},\boldsymbol{a})=\sum_i p(\boldsymbol{z}_i|\boldsymbol{s},\boldsymbol{a})\boldsymbol{z}_i$$

我们从所有的行动中选择使期望最高的行动，可以得到

$$\boldsymbol{a}^*=\text{argmax}_{\boldsymbol{a}}q(\boldsymbol{s},\boldsymbol{a})$$

然后我们可以开始对齐的计算了。从图 8-8 中可以看出，计算过程分为三步。

第 1 步计算出真实的目标价值分布。由于价值采样点是确定的，所以我们可以直接使用公式计算出目标价值的价值点，即进行下面的计算：

$$\boldsymbol{z}_i'=\boldsymbol{r}_t+\gamma_t\boldsymbol{z}_i$$

由于每个采样点是固定的，所以这一步对于每一步更新来说都是相同的。每一个新的价值点的概率和旧的价值点相同，也就是说 $p(\boldsymbol{z}_i'|\boldsymbol{s},\boldsymbol{a}^*)=p(\boldsymbol{z}_i|\boldsymbol{s},\boldsymbol{a}^*)$

第 2 步要完成价值限定的操作。其中落入价值范围之外的值将被强制投影到范围之内。大于最大值的价值采样点会被投影到最大价值处，小于最小值的价值采样点会被投影到最小价值处。

第 3 步要完成价值采样点的投影。最终我们需要把价值采样点 $\boldsymbol{z}_i'$ 投影回原始的采样点 $\boldsymbol{z}_i$。这里的投影方法比较简单，由于第 2 步我们已经将采样点 $\boldsymbol{z}_i'$ 限定在价值范围内，所以就可以找到离采样点 $\boldsymbol{z}_i'$ 最近的两个原始采样点 $\boldsymbol{z}_i$，那么下一步就可以以采样点到原始采样点的距离为权重将采样点 $\boldsymbol{z}_i'$ 的概率分配到原始采样点上，这样就得到了基于原始采样点的目标概率。

再通过模型得到对当前时刻的分布估计，就可以使用 Cross Entropy Loss 求出目标函数了。完整的更新算法如下所示：

算法 Distributional DQN 模型更新

Input

$\boldsymbol{x}_t,\boldsymbol{a}_t,\boldsymbol{r}_t,\boldsymbol{x}_{t+1},\gamma_t\in[0,1]$

Start

第一步：选择最优行动

$q(\boldsymbol{x}_{t+1},\boldsymbol{a})=\sum_i \boldsymbol{z}_i p(\boldsymbol{x}_{t+1},\boldsymbol{a})$, $\boldsymbol{a}^*\leftarrow\text{argmax}_{\boldsymbol{a}}q(\boldsymbol{x}_{t+1},\boldsymbol{a})$

第二步：计算对齐的概率

$\{p'\}_{i=0}^N=0$

for $i=0$ to $N-1$, do:

書籍是人类进步的阶梯

$\boldsymbol{z}_i' \leftarrow [\boldsymbol{r}_t + \gamma \boldsymbol{z}_i]|_{V_{\text{MIN}}}^{V_{\text{MAX}}}$
$b_i \leftarrow (\boldsymbol{z}_i' - V_{\min})/\Delta \boldsymbol{z}$
$l \leftarrow \text{floor}(b_i)$，$u \leftarrow \text{ceil}(b_i)$
$p_l' \leftarrow p_l' + p(\boldsymbol{z}_i'|\boldsymbol{x}_{t+1}, \boldsymbol{a}^*)(u - b_i)$
$p_u' \leftarrow p_u' + p(\boldsymbol{z}_i'|\boldsymbol{x}_{t+1}, \boldsymbol{a}^*)(b_i - l)$
end for
第三步：计算目标函数
$\text{Loss} = -\sum_i p_i' \log p_i(\boldsymbol{x}_t, \boldsymbol{a}_t)$

通过使用分布的形式表示值函数后，模型可以学习到更丰富的信息，这样可以使模型做出更细致的决策。

8.6 Noisy Network

增强探索能力是强化学习中经常遇到的问题，在前面我们曾接触过 $\epsilon-$greedy 的方法，这个算法中以一定的概率 ϵ 随机执行行动，而在剩下的 $1-\epsilon$ 中执行最优行动，这相当于在执行策略的环节增加一定的噪声，使得模型具备一定的探索能力。本节介绍另一种增强探索能力的方法：Noisy Network，来自论文 *Noisy Networks for Exploration*[6]。不同于 $\epsilon-$greedy 的方法，它使用了一种更平滑的手段增加探索能力。

那么，算法是如何实现想要的效果的呢？我们以一个简单的函数为例，来看看它的效果。这个函数的形式为

$$y = wx + b$$

其中 x 表示输入、y 表示输出，w 和 b 是函数的参数。也就是说，如果输入是 x，那么经过函数的变换，输出的结果就会变成 y。这样，我们就可以使用这个函数表示自然界中一组 x 和 y 的关系。当然，自然界中存在着一定的噪声，我们无法直接使用这个函数进行表示，于是给函数加入一个噪声项，于是函数变为

$$y = wx + b + \epsilon$$

其中 ϵ 服从均值为 0，方差为 σ^2 的高斯分布

$$\epsilon \sim N(0, \sigma^2)$$

σ 是一个固定值，表示噪声带来的方差。这样我们也可以认为 y 服从如下的高斯分布：

书籍是人类进步的阶梯

$$y \sim N(wx + b, \sigma^2)$$

可以看出由于噪声的存在，我们可以从同一个 x 映射到多个 y，这相当于增加了输出的不确定性。不确定性对于探索来说十分重要，由于不确定性的存在，我们可以选择确定行动之外的其他行动，因此我们发现噪声和探索存在某些类似的特性，我们可以利用噪声增加模型的探索能力。

一种添加噪声的方法是在参数上增加噪声。对于上面函数中的参数 w，我们可以定义参数来自均值为 μ_w，方差为 σ_w 的高斯分布。同理，参数 b 服从均值为 μ_b，方差为 σ_b 的高斯分布，这样函数就变成了下面的形式：

$$\tilde{w} \sim N(\mu_w, \sigma_w)$$
$$\tilde{b} \sim N(\mu_b, \sigma_b)$$
$$y = \tilde{w}x + \tilde{b}$$

这个形式理解起来并不难，前向计算也并不困难，但是反向计算却有点困难，如何将得到的反向梯度传递到高斯分布中的分布参数呢？为了简化计算，我们需要将参数的表现形式做一定的变换，变成固定部分和随机部分的和，形式如下所示：

$$\tilde{w} = \mu_w + \sigma_w \epsilon$$
$$\tilde{b} = \mu_b + \sigma_b \epsilon$$

ϵ 代表参数中的随机部分，它不属于参数，服从确定的统计分布，例如均值为 0，方差为 1 的高斯分布 $N(0, 1)$。在完成采样后，它可以被当成一个常量对待，这样另外两个参数就可以使用前向后向计算优化了。

当然，上面这种方法也只是为模型添加噪声的方法之一，如果利用这种方法增加噪声，那么如果函数原本有 N 个参数，为了实现噪声的效果，我们需要把参数数量增加一倍，对于小型网络来说，使用这样的方法添加噪声是可行的，但对较大的网络来说，增加一倍的网络参数会给计算带来不小的负担。

为了减少噪声参数的数量，我们还可以从函数参数的结构入手。在我们熟悉的全联接运算中，参数 $\boldsymbol{w}$ 一般是一个二维的矩阵，假设它的维度为 $p \times q$，那么我们可以只生成 $p + q$ 个噪声参数，于是对于上述函数中每一个参数，可以用下面这种全新的形式表示：

$$\boldsymbol{w}_{i,j} = \mu_{\boldsymbol{w}}[i, j] + f(\sigma_p[i]) f(\sigma_q[j]) \epsilon$$

$$b_j = \mu_b[j] + f(\sigma_q[j])\epsilon$$

其中参数数值的第一项保持不变，后面一项的表示有些变化。我们将添加的 $p+q$ 个参数分成两部分，一部分的维度为 p，$\sigma_p[i]$ 表示其中的第 i 个噪声参数；另一部分的维度为 q，$\sigma_q[j]$ 表示其中第 j 个噪声参数。这里的 $f(x) = \text{sgn}(x)\sqrt{x}$。通过这样的设定，我们在噪声效果和噪声参数数量两方面得到了很好的平衡。对于更复杂的模型，我们也可以采用类似的方法添加噪声参数来实现对噪声信息的拟合。

完成了对添加噪声基本思想的介绍，下面就来介绍这个噪声的参数如何融入我们已经介绍的 DQN 算法，我们知道基于 Target Network 的 DQN 算法的目标函数公式为

$$L(\theta) = E_{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{r}_t, \boldsymbol{s}_{t+1}) \sim D}[\boldsymbol{r}_t + \gamma \text{max}_{\boldsymbol{a}* \in \boldsymbol{A}} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a}^*; \theta^-) - Q(\boldsymbol{s}_t, \boldsymbol{a}_t; \theta)]^2$$

其中 θ 表示 Behavior Network 的模型参数，θ^- 表示 Target Network 的模型参数。我们可以在值函数中加入一定的噪声，由于噪声会影响最终的价值输出，也会影响最终的行动，于是噪声的大小影响了模型的探索特性，噪声越小表示探索能力越小，噪声越大表示探索能力越大。我们可以为两个模型参数分别加入噪声随机变量 ϵ 和 ϵ^-，以及噪声参数 σ 和 σ^-，此时新的目标函数变为

$$\begin{aligned} L(\theta) = E_{\epsilon^-, \epsilon}[E_{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{r}_t, \boldsymbol{s}_{t+1}) \sim D}[\boldsymbol{r}_t + \gamma \text{max}_{\boldsymbol{a}* \in \boldsymbol{A}} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a}^*, \epsilon^-; \theta^-, \sigma^-) \\ - Q(\boldsymbol{s}_t, \boldsymbol{a}_t, \epsilon; \theta, \sigma)]^2] \end{aligned}$$

在原本的目标函数中噪声项并不存在，因此此时噪声的加入使得目标函数产生了偏差。为了消除这个偏差，我们可以对噪声求期望，由于噪声的期望值为 0，求解期望后目标函数不再有偏，但是模型依然拥有一定的探索能力。对于这个公式的实现方法，读者一定已经非常熟悉了，我们只需要将其以蒙特卡罗的形式计算即可，公式中的两个期望可以同时转化为求平均的操作，这样这个算法就变得可以计算了。

最后，由于噪声的引入，我们需要考虑噪声参数的初始化。根据论文中的介绍，噪声参数为 $p+q$ 个时，令 $N = p \times q$，参数 μ 按照范围为 $[\frac{1}{\sqrt{N}}, +\frac{1}{\sqrt{N}}]$ 的均匀分布进行初始化，参数 σ 将初始化为常量 $\frac{0.4}{\sqrt{N}}$。

Noisy Network 的思想相对简单，我们通过添加噪声为模型赋予探索的特性，与之前的 $\epsilon-$greedy 的探索不同，Noisy Network 的探索方式更平滑，对探索的粒度控制更细腻，因此在实验中它也获得了不错的成绩。

8.7 Rainbow

8.7.1 Rainbow 的模型特点

本节介绍 Rainbow 模型，它是一个集多长所长的模型，来自论文 *Rainbow: Combining Improvements in Deep Reinforcement Learning*[7]。Rainbow 模型的名字让人一头雾水，它的名字并不代表它使用了某个技术，而是一种文艺的表示。由于它集成了很多模型的特色算法，如果把每一个特色算法想象成一种色彩，那么这些特点组合在一起就形成了五彩颜色的物体，而彩虹刚好是由多种颜色组成的，于是模型就有了彩虹这个名字。

那么，Rainbow 模型融合了哪些模型的技术呢？实际上这些技术都在前面的章节中介绍过，下面我们来回顾这些技术。

1. Double Q-Learning

Double Q-Learning 构建了两个结构相同但是参数数值不同的模型：Behavior Network 和 Target Network。在模型更新时，首先由 Behavior Network 选出 $t+1$ 时刻的最优行动，然后由 Target Network 得到估计 $t+1$ 时刻最优行动的目标价值估计。通过将这两个步骤解耦，我们可以减少 Q-Learning 方法对价值过高估计带来的影响，其中的核心公式为

$$L(\theta) = (\boldsymbol{r}_{t+1} + \gamma q_{\theta^-}(\boldsymbol{s}_{t+1}, \text{argmax}_{\boldsymbol{a}'} q_\theta(\boldsymbol{s}_{t+1}, \boldsymbol{a}')) - q_\theta(\boldsymbol{s}_t, \boldsymbol{a}_t))^2$$

2. Prioritized Replay Buffer

Prioritized Replay Buffer 通过对 Replay Buffer 中不同的样本赋予不同的权重，使得模型有更高的概率训练对自己有更多提升的样本上，同时以较低的概率训练对自己提升有限的样本。样本出现的权重和样本采样时的 TD-Error 有关，样本的学习率和更新权重还可以根据参数进行调整。

3. Dueling Networks

Dueling Networks 将状态行动价值模型 $q(\boldsymbol{s}, \boldsymbol{a})$ 分解成状态价值 $v(\boldsymbol{s})$ 和价值优势 $A(\boldsymbol{s}, \boldsymbol{a})$ 两个部分，分解后的两部分具有明确的含义，而这样的分解也同时降低了训练

的难度，其中的核心公式为

$$q(\boldsymbol{s}_t, \boldsymbol{a}_t) = v(\boldsymbol{s}_t) + (A(\boldsymbol{s}_t, \boldsymbol{a}_t) - \frac{1}{|A|}\sum_{\boldsymbol{a}'} A(\boldsymbol{s}_t, \boldsymbol{a}'_t))$$

4. Multi-step Learning

前面介绍的 Q-Learning 大多通过下一时刻的回报和价值估计得到目标价值，这种方法在前期具有学习速度较慢的弱点，为了克服这个弱点，Multi-step Learning 使用了更多步的回报，这样在训练前期目标价值可以估计得更准确，从而加快训练速度，其中的核心公式为

$$q'(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{r}_{t+1} + \gamma \boldsymbol{r}_{t+2} + \cdots + \gamma^{n-1}\boldsymbol{r}_{t+n} + \max_{\boldsymbol{a}} \gamma^n q(\boldsymbol{s}_{t+n+1}, \boldsymbol{a})$$

5. Distributional Network

DQN 网络只输出了期望形式的价值，而对价值缺少更细致的刻画。Distributional Network 的模型结构可以输出价值的分布形式。我们可以设定价值模型可能的输出范围 $[V_{\text{MIN}}, V_{\text{MAX}}]$，并在范围内以直方图的形式表示价值对应的出现概率，这使模型的表现能力有了很大的提升，分布形式模型的表示形式如下所示：

$$\{z_i = V_{\text{MIN}} + i\Delta z : 0 \leqslant i < N\}$$
$$p_i(\boldsymbol{x}, \boldsymbol{a}) = \frac{\mathrm{e}^{\theta_i(\boldsymbol{x}, \boldsymbol{a})}}{\sum_j \mathrm{e}^{\theta_j(\boldsymbol{x}, \boldsymbol{a})}}$$

6. Noisy Network

模型的探索能力一直是一个需要提高的方面。为了更优雅、灵活地提高模型的探索能力，Noisy Network 为模型参数加入了一定的噪声，通过噪声的随机性改变参数的数值，进而改变模型输出的数值，对应的更新公式为

$$L(\theta) = E_{\epsilon^-, \epsilon}[E_{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{r}_t, \boldsymbol{s}_{t+1}) \sim D}[r_t + \gamma \max_{\boldsymbol{a}* \in \boldsymbol{A}} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a}^*, \epsilon^-; \theta^-, \sigma^-) - Q(\boldsymbol{s}_t, \boldsymbol{a}_t, \epsilon; \theta, \sigma)]^2]$$

可以看到，Rainbow 中使用了很多基于 DQN 算法的改进技术，这些技术在本章都已经介绍。如果说 DQN 模型是一个经典算法，那么 Rainbow 就是站在 DQN 肩膀上，集

多家改进算法于一身的又一个经典算法。对算法中 6 个技术不够熟悉的读者可以回到前面的章节复习其中的内容。

8.7.2 Deep Q Network 的实现

前面已经了解了 Rainbow 模型，接下来介绍 Baselines 中关于 DQN 的实现，对应的代码在 baselines/deepq 文件夹中。这个子项目实现了一个类似“Rainbow”的算法，但是它缺少了一些“颜色”，例如 Distributional DQN 结构等。为了灵活地组合 DQN 算法的各种特性，项目中包含一些比较灵活的配置，例如开启或者关闭其中的某个特性，这样我们可以根据需要配置不同的算法。

算法的主体流程可以用如下伪代码表示：

```
1  def deepq():
2      act, train, update_target = build_train()
3      replay_buffer = choose_one(priority_replay_buffer,replay_buffer)
4      for t in range(max_timesteps):
5          set_param_noise()
6          action = act(obs)
7          new_obs, rew, done, _ = env.step(env_action)
8          replay_buffer.add(obs, action, rew, new_obs, float(done))
9          if done:
10             obs = env.reset()
11         if need_train:
12             experience = replay_buffer.sample()
13             td_errors = train(experience)
14             if use_priority_train:
15                 replay_buffer.update(td_errors)
16         if need_update_target:
17             update_target()
```

让我们先来看算法的主要流程。在第 2 行算法完成了几个模型的构建：act 用于和环境交互，train 用于模型训练，update_target 用于将 Behavior Network 的参数同步到 Target Network。第 3 行创建了 Replay Buffer，如果使用 Priority Replay Buffer，那么在后续的训练过程中，需要考虑每个样本的采样权重和更新权重。第 5 行用于初始化噪声，使用噪声网络时它会派上用场。接下来的第 6~10 行实现了 Agent 与环境交互并将交互样本保存至 Replay Buffer 的过程。接下来的第 11~15 行实现了模型更新的过程，第

16～17 行实现了 Target Network 更新的过程。在主体流程中，我们看到了 Priority Replay Buffer 结构。

从上面的流程可以看出，第 2 行的 build_train 方法完成了所有模型的构建，让我们进一步深入这个函数，其中的流程可以用如下伪代码表示（实际代码的顺序与伪代码不同）：

```
1  def build_train():
2      if param_noise:
3          act_f = build_act_with_param_noise()
4      else:
5          act_f = build_act()
6
7      q_tp1 = target_network(obs_tp1)
8      if double_q:
9          best_act = argmax(behavior_network(obs_tp1))
10         q_tp1_best = sum(q_tp1 * one_hot(best_action))
11     else:
12         q_tp1_best = max(q_tp1)
13     q_t_selected_target = reward + gamma * q_tp1_best
14
15     q_t = behavior_network(obs_t)
16     q_t_selected = sum(q_t * one_hot(real_action))
17
18     td_error = q_t_selected - stop_gradient(q_t_selected_target)
19     train = optimizer.minimize(td_error)
20     update_target = target_network.params.assign(behavior_network.params)
21     return act_f, train, update_target
```

上面的函数主要介绍了与 Q-Learning 有关的部分，其中第 7～13 行实现了目标价值的计算，变量 obs_tp1 实际上是时刻为 $t+1$ 的观测值，所以变量名展开为“observation t plus 1”。第 7 行实现了 $t+1$ 时刻的价值估计，接下来如果采用本章介绍的 Double Q-Learning 算法，那么最优行动的选择将由 Behavior Network 选出，反之则直接由 Target Network 选出，其中 one_hot 方法用于选出目标行动的价值。第 13 行目标价值由 t 时刻的回报和 $t+1$ 时刻的价值估计相加得到。第 15～16 行则实现了价值模型的估计值，接下来在第 18～19 行可以计算出 td_error，并以此进行优化。第 20 行完成了两个网络的参数同步，这样主体的模型更新部分就完成了。

最后只剩下第 2～5 行关于策略部分的实现，其中 build_act 方法比较简单，对应的

简化版伪代码如下所示：

```
def build_act():
    random_actions = rand(0, num_action)
    deterministic_actions = argmax(behavior_network(obs))
    if rand() < eps:
        act = random_actions
    else:
        act = deterministic_actions
    if need_update_eps:
        eps = new_eps
    return act
```

这部分的代码比较简单，实际上实现了 $\epsilon-$greedy 算法，同时拥有更新 ϵ 的功能，在实际的代码中，我们可以选择使用 $\epsilon-$greedy 算法或者确定的算法。而另外一个方法 build_act_with_param_noise 就比较复杂了，它用到了 Noisy DQN 的特性，对应的伪代码如下所示：

```
def build_act_with_param_noise():
    random_actions = rand(0, num_action)
    q_values_perturbed = behavior_noise_network(ob)
    deterministic_action = argmax(q_values_perturbed)
    if rand() < eps:
        act = random_actions
    else:
        act = deterministic_actions

    if need_update_eps:
        eps = new_eps
    if need_reset_noise:
        behavior_noise_network.params.assign(
            behavior_network.params + rand(noise_scale))
    if need_update_kl_threshold:
        kl_threshold = new_kl_threshold
    if need_update_noise_scale:
        behavior_network_adaptive.params.assign(
            behavior_network.params + rand(noise_scale))
        q_prob = softmax(behavior_network(ob))
        q_prob_adaptive = softmax(behavior_network_adaptive(ob))
```

```
22          mean_kl = mean(kl(q_prob, q_prob_adaptive))
23          if mean_kl < kl_threshold:
24              noise_scale *= 1.01
25          else:
26              noise_scale /= 1.01
27      return act
```

这个函数的第 2~8 行代码与前一个函数类似，不同的是这里使用了噪声网络。噪声网络的产生就像前面介绍的那样，使用 Behavior Network 的参数加上噪声得到。从第 12 行起，就需要对噪声的过程做更多的控制。第 12~14 行完成了对参数的噪声的重置，每一个序列结束后，参数的噪声都会重新生成。第 15~16 行用于设置衡量噪声范围指标的上限。这里采用 KL 散度对噪声影响力进行测定，在接下来的代码中可以看到衡量指标计算的全过程。

首先，我们将使用相同的噪声设定生成另一个模型 behavior_network_adaptive，并使用这个模型得到的价值作为估计价值。我们可以认为行动对应的价值估计越高，行动产生的概率就越高，于是价值向量可以作为输入经过 Softmax 层的计算，将价值转换成行动对应的概率。这样没有加入噪声的模型可以得到行动概率 q_prob，而加入了噪声的模型 behavior_network_adaptive 计算得到了行动概率 q_prob_adaptive，得到了这两个概率分布，就可以使用 KL 散度计算两者的距离。最后我们根据 KL 散度值进行判断，如果值大于上限，那么噪声范围就应该适当缩小；如果值小于上限，那么噪声范围就应该适当扩大。

上面的算法没有提到价值模型的具体形式，在本项目中我们可以选择经典的 DQN 实现，也可以选择 Dueling DQN 的实现。DQN 的实现相对简单，而 Dueling DQN 则是在 DQN 的基础上增加一些操作完成的，两个模型的具体结果如图 8-9 所示。

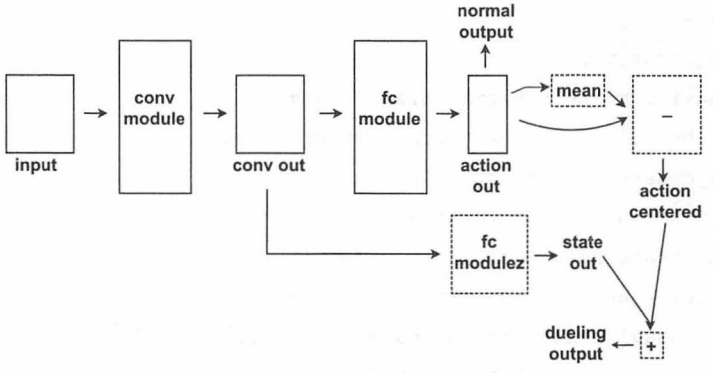


图 8-9　DQN 和 Dueling DQN 模型

这样，我们就了解了这个项目的主要流程，剩下的内容相对比较简单，希望读者自行了解。

8.8 总结

本章介绍了关于 DQN 模型的一些改进算法，并介绍了集成众多算法的 Rainbow 模型，让我们回顾一下。

（1）Priority Replay Buffer 可以更高效地使用其中的样本，让模型更多地选择能让模型提高的样本。

（2）Dueling DQN 将值函数拆解成了两个部分，使模型更容易训练的同时，可以表达更多有价值的信息。

（3）DQN from Demonstration 解决了 Q-Learning 的冷启动问题，也平衡了监督学习和强化学习两部分对模型的影响力。

（4）Distributional DQN 将模型的输出从计算价值期望改为计算价值分布，使模型可以学习更多有价值的信息。

（5）Noisy DQN 通过给参数增加噪声的方式为模型增加一定的探索能力，这种方式更具可控性。

（6）Rainbow 模型集成了上述模型的优势，给出了多种特性融合的可能性。

8.9 参考资料

[1] Van Hasselt H, Guez A, Silver D. Deep Reinforcement Learning with Double Q-learning[J]. Computer Science, 2015.

[2] Schaul T, Quan J, Antonoglou I, et al. Prioritized Experience Replay[J]. Computer Science, 2015.

[3] Wang Z, Schaul T, Hessel M, et al. Dueling network architectures for deep reinforcement learning[J]. 2015:1995-2003.

[4] Hester T, Vecerik M, Pietquin O, et al. Deep Q-learning from Demonstrations[J]. 2018.

[5] Bellemare M G, Dabney W, Munos R. A Distributional Perspective on Reinforcement Learning[J]. 2017.

[6] Fortunato M, Azar M G, Piot B, et al. Noisy Networks for Exploration[J]. 2017.

[7] Hessel M, Modayil J, Van Hasselt H, et al. Rainbow: Combining Improvements in Deep Reinforcement Learning[J]. 2017.

仅供非商业用途或交流学习使用

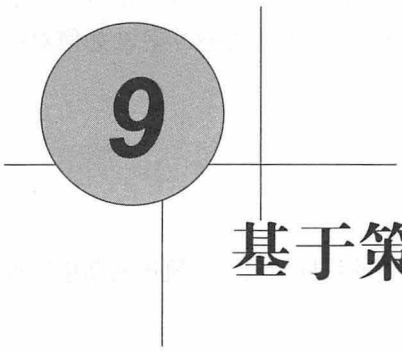
第三部分
基于策略梯度的算法

第三部分将介绍基于策略梯度的强化学习算法，不同于第二部分，本部分介绍的算法将直接基于长期回报期望进行优化，使用的也是在机器学习领域常见的基于梯度的优化方法。策略梯度法一直受到广泛关注，基于这种思想的算法也非常多，从其中很多算法身上都能看到其他领域算法的影子。作为一类经典算法，策略梯度法在学习策略模型过程中表现得相对稳定，同时也可以处理离散和连续行动空间的计算，基于策略模型的改进方案也使算法有了很大的提高。

第三部分共包含三章：

- 第 9 章介绍了策略梯度法的基本原理，以及其改进算法 Actor Critic，并介绍了 A2C 算法的实现。
- 第 10 章介绍了使策略单调提升的算法，其中包括置信区域策略优化（TRPO）和近端策略优化（PPO）。
- 第 11 章介绍了提高样本利用率的策略梯度算法，其中包括基于经验回放的 Actor Critic（ACER）和确定策略梯度法（DPG）。

仅供非商业用途或交流学习使用



基于策略梯度的算法

前面的章节中我们介绍了基于最优价值的强化学习算法，这些方法的核心在于对值函数的估计，如果值函数能够很好地估计得到，我们就可以通过最优值函数得到对应的策略。这个思路在实际中得到了很好的应用，但是它也存在一些问题，例如模型训练的稳定性等。这也使得我们去探索其他的方法，例如本章介绍的策略梯度（Policy Gradient）法并不采取迂回的方式更新策略，而是直接计算策略可能更新的方向。本章，我们将介绍策略梯度方法的原理，同时介绍由此扩展得到的方法：Actor Critic 法，最后介绍在实战中获得比较好结果的方法：A3C 与 A2C。

9.1 策略梯度法

本节将介绍策略梯度法，这种算法和前面介绍的算法有很大不同。实际上，第 6 章 ~ 第 8 章介绍的算法都使用了类似的优化思想，优化的重点都落在了值函数上，无论是策略迭代，还是 Q-Learning，只要能够得到精确的值函数，就可以使用 Bellman 公式求出最优策略，也就是说，我们的最优策略 π^* 需要满足

$$\boldsymbol{a}^* = \text{argmax}_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a}),\ \boldsymbol{a}^* = \pi^*(\boldsymbol{s})$$

策略梯度法则使用了另外一种思路，这个思路实际上更容易被读者理解。让我们回到问题的本源，强化学习的目标是最大化长期回报期望，于是目标也可以写作

$$\pi^* = \text{argmax}_{\pi} E_{\boldsymbol{\tau} \sim \pi(\boldsymbol{\tau})}[r(\boldsymbol{\tau})]$$

仅供非商业用途或交流学习使用

其中 τ 表示使用策略进行交互得到的一条轨迹，$r(\tau)$ 表示这条轨迹的总体回报。由于值函数实际上也是一个函数，强化学习的目标是最大化这个函数，如果这个函数性质良好，我们是不是可以用求导的方法对其进行优化呢？如果我们可以将值函数表示为策略参数的某个函数，就可以求出值函数关于策略参数的梯度，并使参数沿着梯度上升的方向更新，也就可以提升策略了。下面就使用优化的思想介绍这个算法如何对策略进行优化。

9.1.1 算法推导

下面开始公式的推导，我们用 $J(\theta)$ 表示前面提到的目标函数，将轨迹的期望回报展开，可以得到

$$J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[r(\tau)] = \int_{\tau\sim\pi_\theta(\tau)} \pi_\theta(\tau) r(\tau) \mathrm{d}\tau$$

下面对公式求导，由于策略函数通常是定义良好的函数，所以求导运算可以和积分运算互换，这样可以得到

$$\nabla_\theta J(\theta) = \int_{\tau\sim\pi_\theta(\tau)} \nabla_\theta \pi_\theta(\tau) r(\tau) \mathrm{d}\tau$$

因为积分的缘故，这个形式并不方便直接计算，于是我们对其做一个变换，这里用到对数求导的基本公式

$$\nabla_x \log y = \frac{1}{y} \nabla_x y$$

经过变换可以得到

$$y \nabla_x \log y = \nabla_x y$$

我们将 y 替换为 $\pi_\theta(\tau)$，将 x 替换为 θ，同时将公式的左右两边互换，就可以得到

$$\nabla_\theta \pi_\theta(\tau) = \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau)$$

带入前面的公式，得到

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_{\tau\sim\pi_\theta(\tau)} \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) \mathrm{d}\tau \\ &= \boldsymbol{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \end{aligned}$$

仅供非商业用途或交流学习使用

求出的梯度虽然看上去形式很简洁，但其中还是有一些不易计算的部分，例如$\nabla_\theta \log \pi_\theta(\boldsymbol{\tau})$。下面就来看看这些内容如何做进一步拆解。前面我们给出了 $\boldsymbol{\tau}$ 的基本形式，假设轨迹的总长度为 T，将其展开可以得到

$$\begin{aligned}\pi(\boldsymbol{\tau}) &= \pi(\boldsymbol{s}_0, \boldsymbol{a}_0, \cdots, \boldsymbol{s}_T, \boldsymbol{a}_T) \\ &= p(\boldsymbol{s}_0) \prod_{t=0}^{T} \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)\end{aligned}$$

对其求导，可以得到

$$\begin{aligned}\nabla_\theta \log \pi(\boldsymbol{\tau}) &= \nabla_\theta \log [p(\boldsymbol{s}_0) \prod_{t=0}^{T} \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)] \\ &= \nabla_\theta [\log p(\boldsymbol{s}_0) + \sum_{t=0}^{T} \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) + \sum_{t=0}^{T} \log p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)] \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)\end{aligned}$$

得到这样的结果，我们就可以暂时停止进一步展开公式的步伐了，因为此时的公式已经和大家常见的最大 log 似然的公式没有区别。最后，我们再使用已经尝试了多次的蒙特卡罗法，将公式中的期望用蒙特卡罗近似的方式进行替换，可以得到求解梯度的“最终”形式：

$$\begin{aligned}\nabla_\theta J(\theta) &= E_{\boldsymbol{\tau} \sim \pi_\theta(\boldsymbol{\tau})} [(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}))(\sum_{t=0}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}))] \\ &= \frac{1}{N} \sum_{i=1}^{N} [\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})(\sum_{t=0}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}))]\end{aligned}$$

完成了对梯度的求解，剩下的工作就是参数更新。所以总结起来 Policy Gradient 方法分为两步：

（1）计算 $\nabla_\theta J(\theta)$。

（2）$\tilde{\theta} = \theta + \alpha \nabla_\theta J(\theta)$。

实际上，策略梯度的推导还有另外一种更复杂的形式，它主要使用了 Bellman 公式表示交互序列的细节，推导如下所示：

$$\nabla_\theta J(\theta) = \nabla_\theta v_\pi(\boldsymbol{s})$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= \nabla_\theta[\sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a})], \forall \boldsymbol{s} \in \boldsymbol{S} \\
&= \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a}) + \pi(\boldsymbol{a}|\boldsymbol{s}) \nabla_\theta q_\pi(\boldsymbol{s},\boldsymbol{a})] \\
&= \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a}) + \pi(\boldsymbol{a}|\boldsymbol{s}) \nabla_\theta \sum_{\boldsymbol{s}',\boldsymbol{r}} p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})(\boldsymbol{r} + \gamma v_\pi(\boldsymbol{s}'))]
\end{aligned}
$$

这里的 $\boldsymbol{s}'$ 表示状态 $\boldsymbol{s}$ 下一时刻得到的状态，下面的 $\boldsymbol{s}''$ 含义类似，继续推导，可以得到

$$
\begin{aligned}
&= \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a}) + \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \gamma p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \nabla_\theta v_\pi(\boldsymbol{s}')] \\
&= \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a}) + \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \gamma p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \\
&\quad \sum_{\boldsymbol{a}'} [\nabla_\theta \pi(\boldsymbol{a}'|\boldsymbol{s}') q_\pi(\boldsymbol{s}',\boldsymbol{a}') + \pi(\boldsymbol{a}'|\boldsymbol{s}') \nabla_\theta q_\pi(\boldsymbol{s}',\boldsymbol{a}')]] \\
&= \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a}) + \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \gamma p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \\
&\quad \sum_{\boldsymbol{a}'} [\nabla_\theta \pi(\boldsymbol{a}'|\boldsymbol{s}') q_\pi(\boldsymbol{s}',\boldsymbol{a}') + \pi(\boldsymbol{a}'|\boldsymbol{s}') \sum_{\boldsymbol{s}''} \gamma p(\boldsymbol{s}''|\boldsymbol{s}',\boldsymbol{a}') \nabla_\theta v_\pi(\boldsymbol{s}'')]]
\end{aligned}
$$

将公式整理，可以得到

$$
\begin{aligned}
&= \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a})] + \sum_{a} [\pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \gamma p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \sum_{\boldsymbol{a}'} [\nabla_\theta \pi(\boldsymbol{a}'|\boldsymbol{s}') q_\pi(\boldsymbol{s}',\boldsymbol{a}')]] \\
&\quad + \sum_{\boldsymbol{a}} [\pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \gamma p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \sum_{\boldsymbol{a}'} \pi(\boldsymbol{a}'|\boldsymbol{s}') \sum_{\boldsymbol{s}''} \gamma p(\boldsymbol{s}''|\boldsymbol{s}',\boldsymbol{a}') \nabla_\theta v_\pi(\boldsymbol{s}'')]]
\end{aligned}
$$

为了化简公式，我们定义 $Pr(\boldsymbol{s} \to \boldsymbol{x}, k, \pi)$ 为经过 k 时刻环境从状态 $\boldsymbol{s}$ 到状态 $\boldsymbol{x}$ 的概率，那么公式就变为

$$
\begin{aligned}
&= Pr(\boldsymbol{s} \to \boldsymbol{s}, 0, \pi) \sum_{a} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s},\boldsymbol{a})] \\
&\quad + \gamma \sum_{\boldsymbol{s}'} Pr(\boldsymbol{s} \to \boldsymbol{s}', 1, \pi) \sum_{\boldsymbol{a}'} [\nabla_\theta \pi(\boldsymbol{a}'|\boldsymbol{s}') q_\pi(\boldsymbol{s}',\boldsymbol{a}')]] \\
&\quad + \gamma^2 \sum_{\boldsymbol{s}''} Pr(\boldsymbol{s} \to \boldsymbol{s}'', 2, \pi) [\nabla_\theta v_\pi(\boldsymbol{s}'')]
\end{aligned}
$$

将这个公式无限展开，就会变为

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&= Pr(\boldsymbol{s} \to \boldsymbol{s}, 0, \pi) \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s}, \boldsymbol{a})] \\
&\quad + \gamma \sum_{\boldsymbol{s}'} Pr(\boldsymbol{s} \to \boldsymbol{s}', 1, \pi) \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}') q_\pi(\boldsymbol{s}', \boldsymbol{a})]] \\
&\quad + \cdots \\
&\quad + \gamma + ^n \sum_{\boldsymbol{s}^n} Pr(\boldsymbol{s} \to \boldsymbol{s}^n, n, \pi) [\nabla_\theta v_\pi(\boldsymbol{s}^n)]
\end{aligned}
$$

随着 $n \to \infty$，$\gamma^n \to 0$，这个公式展开的项目就会趋近于 0，于是我们将尾项忽略，并将项目的其他部分合并，就可以表示为

$$
= \sum_{\boldsymbol{s}' \in \boldsymbol{S}} \sum_{k=0}^{\infty} \gamma^k Pr(\boldsymbol{s} \to \boldsymbol{s}', k, \pi) \sum_{\boldsymbol{a}} [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s}, \boldsymbol{a})]
$$

其中的

$$
\sum_{k=0}^{\infty} \gamma^k Pr(\boldsymbol{s} \to \boldsymbol{s}', k, \pi)
$$

可以表示某个状态 $\boldsymbol{s}'$ 出现的全部次数的期望，一般可以用变量 $d_\pi(\boldsymbol{s})$ 表示，于是最终公式就变为

$$
= \sum_{\boldsymbol{s}' \in \boldsymbol{S}} \sum_{\boldsymbol{a}} d_\pi(\boldsymbol{s}) [\nabla_\theta \pi(\boldsymbol{a}|\boldsymbol{s}) q_\pi(\boldsymbol{s}, \boldsymbol{a})]
$$

公式的形式就与前面求得的 $E_{\boldsymbol{\tau} \sim \pi_\theta(\boldsymbol{\tau})}[\nabla_\theta \log \pi_\theta(\boldsymbol{\tau}) r(\boldsymbol{\tau})]$ 相同了。

9.1.2 算法分析

由于策略梯度算法的求解形式和我们熟悉的机器学习方法十分相近，所以我们可以从机器学习的角度理解这个方法。其实它和最大似然法十分类似，如果将强化学习问题变成一个监督学习问题，那么我们希望策略模型得到的行动概率分布能够尽可能地与采样数据的行动概率分布一致。我们可以按照这个思路给出模型的求解方法。

首先是准备训练数据，我们将一个序列折开，每一个状态都与下一个状态解耦，同时使用每一步的行动当作模型的目标输出值，这样就把一个交互序列

$$
\boldsymbol{\tau} \sim \{\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{s}_1, \boldsymbol{a}_1, \cdots, \boldsymbol{s}_T, \boldsymbol{a}_T\}
$$

拆解成由状态-行动组成的训练数据集合：

仅供非商业用途或交流学习使用

$$\{(\boldsymbol{s}_0, \boldsymbol{a}_0), (\boldsymbol{s}_1, \boldsymbol{a}_1), \cdots (\boldsymbol{s}_T, \boldsymbol{a}_T)\}$$

这样就可以进行监督学习了。我们采用最大似然法进行建模，目标函数可以设立为

$$\theta^* = \text{argmax}_{\pi_\theta} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \log \pi_\theta(\boldsymbol{a}_{i,t} | \boldsymbol{s}_{i,t})$$

令等号右边的公式为 $J(\theta)$，对其求梯度，可以得到

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t} | \boldsymbol{s}_{i,t})$$

我们将这个公式与前面得到的 Policy Gradient 的梯度进行对比，可以看出两个公式的梯度值十分相近，唯一的不同就是公式后面的一项 $\sum_{t=0}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t})$，这一项表示了序列的整体回报。关于这个差异，我们可以从以下两个角度分析。

（1）从策略梯度法的角度看，最大似然法没有考虑长期回报，或者说最大似然法将长期回报值设定为固定值 1，即每一个样本对训练的影响相同。

（2）从最大似然法的角度看，最大似然对每个样本使用了相同的权重，对策略梯度法来说，它使用序列的回报作为样本的加权权重。也就是说，对于序列回报为正的训练样本，我们要最大化它的似然；对于序列回报为负的训练样本，我们要最小化它的似然。

这样策略梯度算法就和最大似然法在思想上得到了统一，序列回报的功能和样本权重比较类似，我们可以用类似的思路看待这个新事物。

9.1.3 算法改进

看上去我们的算法介绍已经告一段落，可以进入实战阶段，但实际上这个公式还存在一些需要解决的问题。主要问题来自和最大似然法不同的这一部分：序列回报部分。

首先，我们再回顾一下求解策略价值梯度的公式：

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t} | \boldsymbol{s}_{i,t}) (\sum_{t=0}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}))$$

从这个公式中我们看出了一个问题，不论是哪个时间段，我们都要用策略的梯度乘以所有时刻的回报值总和，这样的设计显然是不合理的。因为理论上，在 t 时刻我们

完成了决策后，它最多只能影响 t 时刻之后的所有回报，并不会影响 t 时刻之前的回报，因为我们无法通过现在的决策影响已经发生过的事情，所以这一部分的回报值不应该计算在梯度中。我们的公式应该改写成

$$\nabla_\theta J(\theta)=\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}[\nabla_\theta \log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})(\sum_{t'=t}^{T}r(\boldsymbol{s}_{i,t'},\boldsymbol{a}_{i,t'}))]$$

现在的公式中用两个不同的变量 t 和 t' 表示两部分需要计算的时间，这样公式就变得更合理了。

其次，从上面的分析来看，策略梯度法更像是加权版的最大似然优化法。“权重”将直接影响梯度的更新量，这样就会带来以下两个问题。

（1）如果计算得出的序列回报数值较大，那么对应的参数更新量就会比较大，这样我们的优化就可能出现一定波动，这些波动很可能影响优化效果。

（2）在一些强化学习问题中，环境给予的回报始终为正，那么不论我们的决策如何，最终累积的长期回报值都是一个正数。换句话说，我们会增强所有的策略，只是对于实际效果并不好的策略，我们为其提升的幅度会有所降低。这样的更新方法和我们的初衷并不一致，我们降低不好的行动的概率，而不是轻微提升不好的行动的概率。

这两个问题使我们回到了强化学习的目标：提高能最大化长期回报策略的概率，降低无法最大化长期回报策略的概率。将上面的思想转化成策略梯度问题的表述形式，就会变成：让能够最大化长期回报策略的“权重”为正且尽可能大，让不能最大化长期回报策略的“权重”为负且尽可能小。

为了实现这个目标，我们可以调整权重的数值和范围，一个简单的方法就是给所有时刻的长期累积回报减去一个偏移量，这个偏移量也被称为 Baseline，我们用变量 $\boldsymbol{b}$ 表示，于是公式就变为

$$\nabla_\theta J(\theta)=\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}[\nabla_\theta \log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})(\sum_{t'=t}^{T}r(\boldsymbol{s}_{i,t'},\boldsymbol{a}_{i,t'})-\boldsymbol{b}_{i,t'})]$$

这个变量可以设计为同一起始点的不同序列在同一时刻的长期回报均值，它的公式形式如下所示：

$$\boldsymbol{b}_{i,t'}=\frac{1}{N}\sum_{i=1}^{N}\sum_{t'=t}^{T}r(\boldsymbol{s}_{i,t'},\boldsymbol{a}_{i,t'})$$

经过这样的处理，所有时刻的权重均值变为 0，就会存在权重为正或为负的行动，同时权重的绝对值也得到了一定的缩小，这对于算法的稳定性来说都有一定的帮助。更

重要的是，加入的这个偏移量并不会使原本的计算值变得有偏，我们可以得到

$$
\begin{aligned}
E[\nabla_\theta \log \pi_\theta(\boldsymbol{\tau})\boldsymbol{b}] &= \int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \pi_\theta(\boldsymbol{\tau})\nabla_\theta \log \pi_\theta(\boldsymbol{\tau})\boldsymbol{b}\mathrm{d}\boldsymbol{\tau} \\
&= \int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \nabla_\theta \pi_\theta(\boldsymbol{\tau})\boldsymbol{b}\mathrm{d}\boldsymbol{\tau} \\
&= \boldsymbol{b}\int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \nabla_\theta \pi_\theta(\boldsymbol{\tau})\mathrm{d}\boldsymbol{\tau} \\
&= \boldsymbol{b}\nabla_\theta \int_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})} \pi_\theta(\boldsymbol{\tau})\mathrm{d}\boldsymbol{\tau} \\
&= \boldsymbol{b}\nabla_\theta 1 \\
&= 0
\end{aligned}
$$

所以它可以在不影响期望值的同时降低算法的波动性。这个方法也被广泛应用于 Policy Gradient 算法中。当然，在现在常见的方法中，我们已经有了更好的替代方法，这部分内容将在 9.2 节介绍。

9.2 Actor-Critic 算法

9.2.1 降低算法的方差

在 9.1 节中我们介绍了策略梯度法，这个方法已经能够利用交互的样本进行训练了，但是它还存在着一些问题。我们再来看看下面这个公式：

$$
\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}[\nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})(\sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t'}, \boldsymbol{a}_{i,t'}) - \boldsymbol{b}_i)]
$$

公式的问题仍然在等号右边的最后一项。前面提到，我们用轨迹的回报表示整个序列的价值，这个表示是准确无偏的，但是在真实的训练过程中，为了尽可能地控制学习时间，我们无法完成非常多次的交互，往往只会进行有限次数的交互，这些交互有时并不能很好地代表轨迹真实的期望。每一个交互得到的序列都会有一定的差异，对应的回报也会有一定的差异，因此不充足的交互会给轨迹回报带来较大的方差，这和前面提到的蒙特卡罗法遇到的问题类似，如图 9-1 所示。

由于强化学习中 Agent 与环境交互基于 MDP，每一时刻 Agent 基于当前的状态可以有很多的选择，而采样得到的序列只是其中之一，所以一条或几条序列的回报和所

有路径的回报期望存在一定的差距。

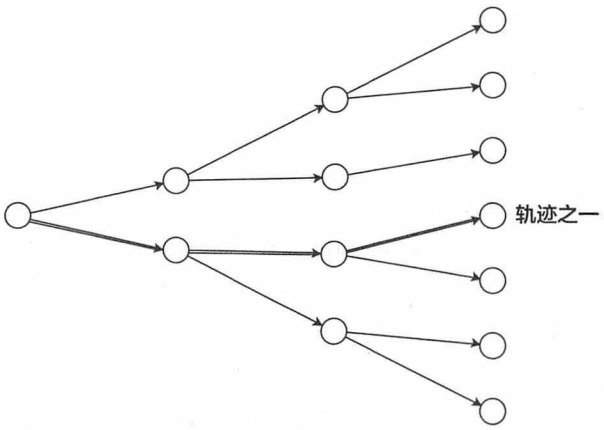


图 9-1 交互序列的差异

机器学习的一个核心问题是平衡偏差和方差。对策略梯度模型来说，它的方差相对较大，为了模型的稳定，我们可以牺牲一定的偏差来换取方差变小。这其中的一种方法就是采用 Actor-Critic 算法，其主要特点就是用一个独立的模型估计轨迹的长期回报，而不再直接使用轨迹的真实回报。类似于基于模型的 Q-Learning 算法，在估计时使用模型估计轨迹价值，在更新时利用轨迹的回报得到目标价值，然后将模型的估计值和目标值进行比较，从而改进模型。

公式中可以被模型替代的部分有两个，其中 $\sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t'}, \boldsymbol{a}_{i,t'})$ 代表从 t 时刻出发所获得的长期回报，$\boldsymbol{b}_i$ 代表待减去的偏移量。根据这两个部分被模型替代的情况，我们可以得到以下几种方案。

（1）采用策略梯度法的方法：公式由 $\sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t'}, \boldsymbol{a}_{i,t'}) - \boldsymbol{b}_i$ 表示。

（2）使用状态值函数估计轨迹的回报：$q(\boldsymbol{s}, \boldsymbol{a})$。

（3）直接使用优势函数估计轨迹的回报：$A(\boldsymbol{s}, \boldsymbol{a}) = q(\boldsymbol{s}, \boldsymbol{a}) - V(\boldsymbol{s})$。

（4）使用 TD-Error 估计轨迹的回报：公式由 $r(\boldsymbol{s}_t, \boldsymbol{a}_t) + v(\boldsymbol{s}_{t+1}) - v(\boldsymbol{s})$ 表示。

可以看出四种方法都可以在一定程度上降低算法的方差，实际中 Actor Critic 算法最终选择了第 4 种方案，这种方案在计算量和减少方差方面具有一定的优势。由于引入了状态价值模型，算法整体包含了两个模型，一个是策略模型，一个是价值模型，所以这个算法被称为 Actor-Critic，其中 Actor 表示策略模型，Critic 表示价值模型。

9.2.2 A3C 算法

真正将 Actor-Critic 应用到实际中并得到优异效果的是 A3C（Asynchronous Advantage Actor-Critic）算法，从算法的名字中可以看出，算法突出了异步和优势两个概念。另一个优化算法 A2C（Advantage Actor-Critic），采用了同步的方法进行优化，并同样获得了比较好的效果。

Asynchronous 表示了 A3C 算法的重要特点之一：并行的交互采样和训练。通过 9.1 节我们已经知道，策略梯度法和 Actor-Critic 算法都通过目标函数的梯度进行策略更新，而计算梯度需要基于当前的策略模型，所以每一次计算梯度时，我们需要使用当前最新的策略模型重新进行交互采样，得到相应的序列样本，然后使用这些样本完成梯度计算；当梯度计算完成，模型得到更新后，这些使用过的样本将被丢弃，我们需要采集新的样本。这种训练方式通常被称为 **On-Policy Learning**，也就是说我们要基于样本进行学习，第 7 章中的 SARSA 算法也是这样的算法。

第 7 章和第 8 章介绍的 DQN 方法，由于采用 Replay Buffer 存储了一段时间的交互样本，模型学习时使用的样本不一定是由当前模型交互得到的，而且 Q-Learning 在学习时假设在 $t+1$ 时刻将采用当前最优的行动，因此它的目标价值并不是由真实的交互序列得到的。这样的学习方式通常被称为 **Off-Policy Learning**。

由于 Off-Policy 算法可以使用非当前策略产生的样本，因此它对样本的重复使用度比较高。而对 On-Policy 算法来说，每一次模型更新都需要一定量的“新样本”，所以 On-Policy 算法需要花费更多的时间收集样本。所以为了更快地收集样本，我们需要用并行的方法收集。在 A3C 方法中，我们要同时启动 N 个线程，Agent 将在 N 个线程中同时进行环境交互。只要保证每一个线程中的环境设定不同，线程间交互得到的序列就不会完全一样，这样得到的样本就很有意义。收集完样本后，每一个线程将独立完成训练并得到参数更新量，并异步地更新到全局的模型参数中。下一次训练时，线程的模型参数将和全局参数完成同步，再使用新的参数进行新一轮的训练。

Advantage 表示了算法关于“权重”部分的计算方法。前面提到 Actor-Critic 算法并不是真的直接使用优势函数表示权重，而是采用 TD-Error 的形式：$r_{s_t,a_t}+v(s_{t+1})-v(s_t)$，公式通过当前时刻的回报和下一时刻的价值估计得到了目标价值，然后减去当前时刻的价值估计。这个方法虽然增加了模型的稳定性，但是模型的偏差也相应变大，为了更好地平衡模型的偏差和方差，A3C 方法使用了 8.4 节介绍的多步回报估计法，这个方法可以在训练早期更快地提升价值模型。对应的公式变为

$$\sum_{i=1}^{n}\gamma^{i-1}r_{t+1}+v(s_{t+n})-v(s_t)$$

最后，为了增加模型的探索性，模型的目标函数中加入了策略的熵。由于熵可以衡量概率分布的不确定性，所以我们希望模型的熵尽可能大一些，这样模型就可以拥有更好的多样性。这样，完整的策略梯度计算公式就变为

$$\nabla_\theta J(\theta)=\frac{1}{T}\sum_{t}^{T}\nabla_\theta \log\pi(\boldsymbol{a}_t|\boldsymbol{s}_t;\theta)(\sum_{i=1}^{n}\gamma^{i-1}\boldsymbol{r}_{t+1}+v(\boldsymbol{s}_{t+n})-v(\boldsymbol{s}_t))+\beta\nabla_\theta \mathrm{H}(\pi(\boldsymbol{s}_t;\theta))$$

其中 β 为策略的熵在目标中的权重。价值模型的目标函数与 DQN 类似，这里不再赘述。A3C 的完整算法如下所示：

A3C 算法 单一线程执行过程

Input

$T\leftarrow 0$

全局和线程内部的策略模型参数：θ,θ'

全局和线程内部的价值模型参数：θ_v,θ'_v

两个模型的梯度更新量：$\mathrm{d}\theta,\mathrm{d}\theta_v$

Start

repeat until $T>T_{\max}$

将梯度清零：$\mathrm{d}\theta\leftarrow 0,\mathrm{d}\theta_v\leftarrow 0$

同步模型参数：$\theta'\leftarrow\theta$，$\theta'_v\leftarrow\theta_v$

用策略模型 $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t;\theta')$ 完成一次轨迹采样，得到 $\{\boldsymbol{s}_0,\boldsymbol{a}_0,\boldsymbol{r}_0,\ldots\}$

$T\leftarrow T+n$

计算每一个采样的价值：

$$R=\begin{cases}0 & \text{对于终点状态}\\ v(\boldsymbol{s}_t;\theta'_v) & \text{对于非终点状态}\end{cases}$$

for $i\in\{n-1,0\}$ do

$R\leftarrow \boldsymbol{r}_i+\gamma R$

积累策略模型梯度 θ'：$\mathrm{d}\theta\leftarrow\mathrm{d}\theta+\nabla_{\theta'}\log\pi(\boldsymbol{a}_i|\boldsymbol{s}_i;\theta')(R-V(\boldsymbol{s}_i;\theta'_v))$

积累价值模型梯度 θ'_v：$\mathrm{d}\theta_v\leftarrow\mathrm{d}\theta_v+\nabla_{\theta'_v}(R-V(\boldsymbol{s}_i;\theta'_v))^2$

end for

对全局模型 θ 和 θ_v 进行异步更新：$\theta\leftarrow\mathrm{d}\theta$，$\theta_v\leftarrow\mathrm{d}\theta_v$

由于其十分优异的效果，A3C 算法的影响力极大。但是大家一直存在一个疑问：算法中的异步更新是否是必要的？

算法作者的实验环境和我们常见的基于 GPU 的环境不同，它主要使用了 CPU 进行模拟和训练，并且使用了“Hogwild!”[3] 这样的更新方法，这和深度学习中常见的基

于同步更新的方法并不一样。凭直觉，异步或者同步更新并不是决定算法优劣的主要因素，那么为什么不尝试使用同步更新方法呢？于是大家开始尝试非异步的方法，这就是 A2C 的来源。A3C 和 A2C 更新流程的对比如图 9-2 所示。可以看出，在 A3C 中，每个线程各自完成参数更新值的计算，并异步完成参数的同步操作；而 A2C 将决策和训练的任务集中到一处，其他进程只负责环境模拟的工作。

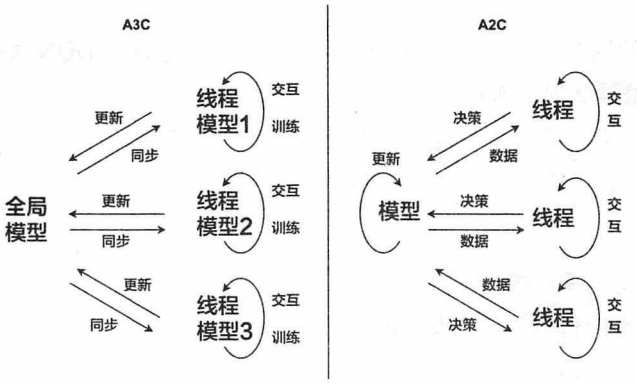


图 9-2　A3C 与 A2C 的更新区别

在 Baseline 项目中，我们可以看到 A2C 的实现，而且 OpenAI 的官方博客中也提到 [5] A2C 的效果要优于 A3C，而且 A2C 在并发性和系统简洁性上都优于 A3C。那么我们就来介绍 A2C 的实现细节。

9.2.3　A2C 算法实战

本节我们介绍 A2C 算法在实战中的效果。A2C 的代码在 Baselines 项目中的 baselines/a2c 文件夹，读者可以直接执行其中的 run_atari.py 文件执行算法。

首先我们介绍算法与环境交互的实现。这个算法的代码采用了 Master-Slave 的结构，其中的 Master 用于执行 Agent 的模型，在 Env 给出状态观测值后判断应该执行的 Action；而 Slave 用于模拟 Env，在收到 Agent 的行动后，将后续的观测值、回报等信息返回给 Master，Master 将 Slave 接收不了的数据收集并进行训练。这个架构比较简洁清晰，代码中的 Master-Slave 结构如图 9-3 所示。

然后我们介绍算法的模型结构。在代码的 policies.py 文件中，我们可以看到三个表示策略的模型：LnLstmPolicy、LstmPolicy 和 CnnPolicy。从名字上可以看出，它们采用了不同的模型结构，其中 LnLstm 代表 Layer Normalization LSTM。三个模型使用了相同的卷积神经网络结构 Nature CNN 用于从 Atari 的游戏画面中提取特征，这个网络名

字来源于 DQN 的网络结构，因为发明 DQN 模型的论文发表在 Nature 上。Nature CNN 的网络结构如图 9-4 所示。

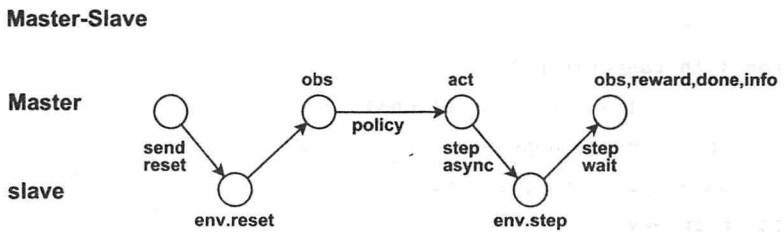


图 9-3 Master-Slave 的结构

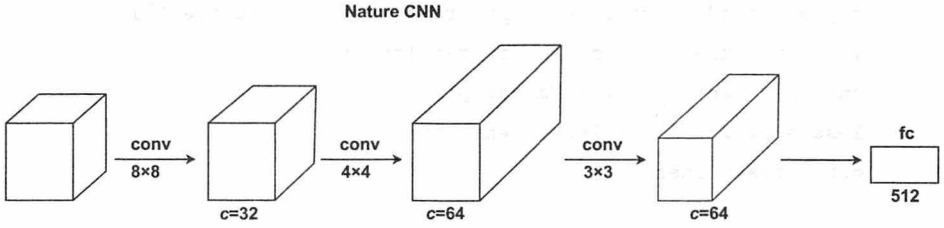


图 9-4 Nature CNN 的网络结构

CNNPolicy 的结构相对简单，使用 Nature CNN 提取特征后再经过全连接层的计算，就可以得到观测值的行动概率和价值，其结构如图 9-5 所示。

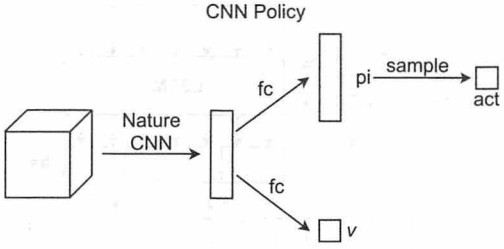


图 9-5 CNNPolicy 的网络结构

LstmPolicy 则相对复杂，除了使用 Nature CNN 提取特征，我们将时长为 T 的特征转换成序列的形式，并通过 LSTM 得到转换后的序列特征，再经过全连接层的处理，就可以得到行动概率和价值，其结构如图 9-6 所示。

LnLstmPolicy 的结构和 LstmPolicy 类似，这里就不再介绍。了解了这两部分，我们就可以看看模型的核心部分了，主体训练流程如下面的伪代码所示：

```
def learn():
    for i in batches:
        # run
        obs = env.reset()
        for j in range(steps):
            act, value = step_model(obs)
            obs, reward, done = env.step(act)
            data.append(obs, reward, done, value)
        for n in envs:
            data[reward] = discount_target_reward(data[reward], value)
        # train
        adv = data[reward] - data[value]
        pg_loss = adv * cross_entropy(train_model.pi, data[act])
        vf_loss = mse(train_model.vf, data[reward])
        entropy = entropy(train_model.pi)
        loss = pg_loss + vf_loss - entropy
        opt = train(loss)
```

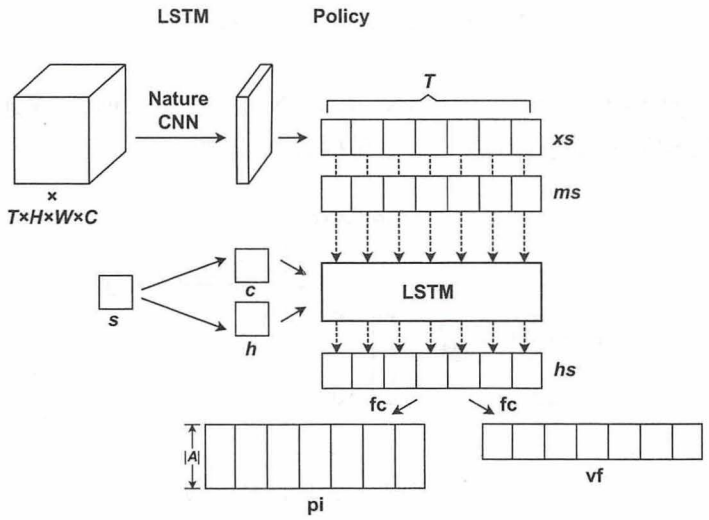


图 9-6　LstmPolicy 的网络结构

从伪代码中可以看出，整体流程分为两个部分：与环境交互和基于样本训练模型。在与环境交互时，我们使用 step_model 进行交互，这个模型接收 N 个 Slave 发来的观测值，并给出单步的行动决策。为了方便计算，每一个轨迹只选取前 steps 进行交互，交互完成后，这一时刻的 obs、reward 和这一时刻的价值估计 value 将被保存。当数据收

书籍是人类进步的阶梯

集完成后，下一步就是计算每一时刻的目标价值估计值，并更新到 data[reward] 中。

接下来就是训练过程，代码中分别计算了三部分的目标函数。

- 策略 Loss：$\sum \text{Adv} \times \log\pi(\boldsymbol{a}|\boldsymbol{s})$。
- 价值模型 Loss：$\frac{1}{2}\sum_i (V(\boldsymbol{s}_i) - R(\boldsymbol{s}_i))^2$。
- 策略的熵 Loss：$-\sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s})\log\pi(\boldsymbol{a}|\boldsymbol{s})$。

由于代码中实际上要完成梯度下降的操作，所以每一项的符号和前面概念中介绍的刚好相反。得到了目标值后，我们直接使用 TensorFlow 的反向计算就可以完成梯度更新。以上就是 A2C 算法的流程，其中更多细节内容欢迎读者自行阅读。

9.3 总结

本章介绍了基于策略梯度的算法：策略梯度法和 Actor Critic 法，让我们回顾一下。

（1）策略梯度法的主要思想是直接计算目标函数关于策略的梯度，通过梯度优化策略。

（2）在策略梯度法中，轨迹的回报起到了类似样本权重的作用。

（3）Actor Critic 法引入了价值模型，并使用 TD-Error 的计算结果作为样本权重。

（4）A3C 方法使用了并发收集交互样本的方式快速收集样本，并采用异步更新参数的方式。

9.4 参考资料

[1] Sutton R S. Policy Gradient Methods for Reinforcement Learning with Function Approximation[J]. Submitted to Advances in Neural Information Processing Systems, 1999, 12:1057-1063.

[2] Konda V R, Tsitsiklis J N. On Actor-Critic Algorithms[M]. Society for Industrial and Applied Mathematics, 2003.

[3] Feng N, Recht B, Re C, et al. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent[J]. Advances in Neural Information Processing Systems, 2011, 24:693-701.

[4] Mnih V, Badia A P, Mirza M, et al. Asynchronous Methods for Deep Reinforcement Learning[J]. 2016.

书籍是人类进步的阶梯

10 使策略单调提升的优化算法

在第 7 章和第 8 章中，我们了解了 DQN 算法，这个算法具有很多很好的特性，但是在训练时模型存在一定的波动性；第 9 章介绍的 A3C 算法平衡了模型方差和偏差，使得模型能够尽量减少波动，但波动性的问题依然存在。于是我们不禁在想：能不能找到一种算法尽可能地减少波动？能不能找到一种训练方法，使模型在保持前面各种优点的同时，效果能稳步上升呢？本章介绍的算法会给我们一些启示。

第 3 章已经了解了自然梯度法这个优化方法，本章介绍以此为基础的方法：置信区域策略优化（Trust Region Policy Optimization，TRPO）和近端策略优化（Proximal Policy Optimization，PPO）方法，它们的总体优化目标比较相近，主要区别在于具体的计算方法。本章的涉及大量的数学知识，请读者根据自身的能力进行学习。

10.1 TRPO

TRPO 算法来自论文 *Trust Region Policy Optimization*[1]。TRPO 算法的计算过程比较复杂，但是它可以确保策略模型在优化时单调提升。为了达到算法想要的效果，我们需要进行一系列的分析和推导，其中主要的思路是找到一种衡量策略之间优劣的计算方法，并以此为目标最大化新策略与旧策略相比的优势。虽然算法的整体思路比较简单，但想要完成这些计算和证明却有些困难，让我们一步一步完成目标公式的推导与求解。

书籍是人类进步的阶梯

10.1.1 策略的差距

TRPO 算法的目标是确保每一轮更新后策略算法都能够有进步，或者说至少不退步。为了确保这一点能够成立，我们首先要得出衡量策略间效果对比的公式，这样才能进行后面的计算。我们先定义基于某个策略的期望价值：

$$\eta(\pi) = E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\boldsymbol{s}_1,\ldots}[\sum_{t=0}^{\infty}\gamma^t r(\boldsymbol{s}_t)]$$

其中 $\boldsymbol{s}_0 \sim \rho_0(\boldsymbol{s}_0)$，$\boldsymbol{a}_t \sim \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$，$\boldsymbol{s}_{t+1} \sim P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t,\boldsymbol{a}_t)$，公式中的变量相信读者已经比较熟悉。接下来给出状态行动值函数、状态值函数和优势函数的定义：

$$Q_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t) = E_{\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1},\ldots}[\sum_{l=0}^{\infty}\gamma^l r(\boldsymbol{s}_{t+l})]$$

$$V_\pi(\boldsymbol{s}_t) = E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[\sum_{l=0}^{\infty}\gamma^l r(\boldsymbol{s}_{t+l})]$$

$$A_\pi(\boldsymbol{s},\boldsymbol{a}) = Q_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t) - V_\pi(\boldsymbol{s}_t)$$

通过上面的定义，我们可以给出表示两个策略的期望价值差距的公式：

$$\eta(\tilde{\pi}) = \eta(\pi) + E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t A_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)]$$

它的证明过程如下，由于

$$\begin{aligned}A_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t) &= Q_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t) - V_\pi(\boldsymbol{s}_t)\\ &= E_{\boldsymbol{s}_{t+1}\sim P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t,\boldsymbol{a}_t)}[r(\boldsymbol{s}_t) + \gamma V_\pi(\boldsymbol{s}_{t+1}) - V_\pi(\boldsymbol{s}_t)]\end{aligned}$$

我们可以得到

$$\begin{aligned}&E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t A_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)]\\ &= E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t E_{\boldsymbol{s}'\sim P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t,\boldsymbol{a}_t)}[r(\boldsymbol{s}_t) + \gamma V_\pi(\boldsymbol{s}_{t+1}) - V_\pi(\boldsymbol{s}_t)]]\\ &= E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}E_{\boldsymbol{s}'\sim P(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t,\boldsymbol{a}_t)}[\sum_{t=0}^{\infty}\gamma^t(r(\boldsymbol{s}_t) + \gamma V_\pi(\boldsymbol{s}_{t+1}) - V_\pi(\boldsymbol{s}_t))]\end{aligned}$$

书籍是人类进步的阶梯

$$
\begin{aligned}
&= E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t(r(\boldsymbol{s}_t)+\gamma V_\pi(\boldsymbol{s}_{t+1})-V_\pi(\boldsymbol{s}_t))]\\
&= E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t r(\boldsymbol{s}_t)]+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t(\gamma V_\pi(\boldsymbol{s}_{t+1})-V_\pi(\boldsymbol{s}_t))]\\
&= \eta(\tilde{\pi})+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t(\gamma E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[\sum_{l=1}^{\infty}\gamma^l r(\boldsymbol{s}_{t+l})]-E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[\sum_{l=0}^{\infty}\gamma^l r(\boldsymbol{s}_{t+l})])]\\
&= \eta(\tilde{\pi})+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t(E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[\sum_{l=1}^{\infty}\gamma^{l+1} r(\boldsymbol{s}_{t+l})]-E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[\sum_{l=0}^{\infty}\gamma^l r(\boldsymbol{s}_{t+l})])]\\
&= \eta(\tilde{\pi})+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t(E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[\sum_{l=1}^{\infty}\gamma^{l+1} r(\boldsymbol{s}_{t+l})]-\sum_{l=0}^{\infty}\gamma^l r(\boldsymbol{s}_{t+l})])]\\
&= \eta(\tilde{\pi})+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[\sum_{t=0}^{\infty}\gamma^t(E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[-r(\boldsymbol{s}_t)])]\\
&= \eta(\tilde{\pi})+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[E_{\boldsymbol{a}_t,\boldsymbol{s}_{t+1},\ldots}[-\sum_{t=0}^{\infty}\gamma^t r(\boldsymbol{s}_t)]]\\
&= \eta(\tilde{\pi})+E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\tilde{\pi}}[-\eta(\pi)]\\
&= \eta(\tilde{\pi})-\eta(\pi)
\end{aligned}
$$

将等式左右进行变换，就可以得到最终结果。

经过前面的推导，我们找到了两个策略差距的基本形式。上面的公式并不能直接计算，我们还要对公式做进一步变换。

令 $\rho_\pi(\boldsymbol{s})$ 表示任意时刻状态 $\boldsymbol{s}$ 的访问概率和，它的形式为

$$
\begin{aligned}
\rho_\pi(\boldsymbol{s}) &= P(\boldsymbol{s}_0=\boldsymbol{s})+\gamma P(\boldsymbol{s}_1=\boldsymbol{s})+\gamma^2 P(\boldsymbol{s}_2=\boldsymbol{s})+\cdots\\
&= \sum_{t=0}^{\infty}\gamma^t P(\boldsymbol{s}_t=\boldsymbol{s})
\end{aligned}
$$

上面的公式可以变形为

$$
\begin{aligned}
\eta(\tilde{\pi}) &= \eta(\pi)+\sum_{t=0}^{\infty}\sum_{\boldsymbol{s}}P(\boldsymbol{s}_t=\boldsymbol{s}|\tilde{\pi})\sum_{\boldsymbol{a}}\tilde{\pi}(\boldsymbol{a}|\boldsymbol{s})\gamma^t A_\pi(\boldsymbol{s},\boldsymbol{a})\\
&= \eta(\pi)+\sum_{\boldsymbol{s}}\sum_{t=0}^{\infty}P(\boldsymbol{s}_t=\boldsymbol{s}|\tilde{\pi})\sum_{\boldsymbol{a}}\tilde{\pi}(\boldsymbol{a}|\boldsymbol{s})\gamma^t A_\pi(\boldsymbol{s},\boldsymbol{a})\\
&= \eta(\pi)+\sum_{\boldsymbol{s}}\sum_{t=0}^{\infty}\gamma^t P(\boldsymbol{s}_t=\boldsymbol{s}|\tilde{\pi})\sum_{\boldsymbol{a}}\tilde{\pi}(\boldsymbol{a}|\boldsymbol{s})A_\pi(\boldsymbol{s},\boldsymbol{a})
\end{aligned}
$$

书籍是人类进步的阶梯

$$= \eta(\pi) + \sum_{s} \rho_{\tilde{\pi}(s)} \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s,a)$$

这样我们就求出了策略之间的差距。从这个公式可以看出，当

$$\sum_{s} \rho_{\tilde{\pi}(s)} \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s,a)$$

保持非负值，可以得到 $\eta(\pi) \leqslant \eta(\tilde{\pi})$。也就是说，我们从某个策略 π_0 出发，通过计算找到一个策略 π_1，使得

$$\sum_{s} \rho_{\pi_1(s)} \sum_{a} \pi_1(a|s) A_{\pi_0}(s,a) \geqslant 0$$

那么我们可以确定策略 π_1 在总体上优于 π_0，依此类推，我们可以不断地找到这样的策略，使策略模型的效果不断增强，直至达到目标，这就是算法单调提升的基本原理。

10.1.2 策略提升的目标公式

10.1.1 节得到了策略提升的目标，但是受目标公式所限，这样寻找策略的方法在实际中是几乎不可行的。因为公式中包含 $\rho_{\tilde{\pi}}(s)$，也就是说，我们需要按照新策略与环境交互得到状态的访问频率，而这个新策略正是我们要求解的策略。所以想要求解这个策略，需要先确定新策略的形式，然后使用这个策略得到一定量的样本，并最终通过这些样本统计判断这个策略是否满足需求。这样我们就需要不断地尝试验证每一个可能的新策略，更新过程就会慢到难以忍受，变得在实际问题中无法执行。

因此，为了让计算变得可行，我们需要找到与上面公式的近似且可解的形式，于是等式被近似为下面这个公式：

$$L_{\pi}(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_{\pi}(s) \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s,a)$$

可以看到，前后两个公式唯一的差别在状态访问频率上。前面的公式需要知道基于新策略的状态访问概率 $\rho_{\tilde{\pi}}(s)$ 进行采样，后面的公式只需要基于旧策略 $\rho_{\tilde{\pi}}(s)$ 进行采样即可，这样一来，在寻找新策略时这一项已经变成了一个已知项，公式变得相对容易求解。那么这个公式能否满足我们的需求呢？

实际上通过证明我们发现，对当前策略 π 来说，两个公式在数值和导数上是相同的：

书籍是人类进步的阶梯

$$L_{\pi_{\theta_0}}(\pi_{\theta_0}) = \eta(\pi_{\theta_0})$$
$$\nabla_\theta L_{\pi_{\theta_0}}(\pi_\theta)|_{\theta=\theta_0} = \nabla_\theta \eta(\pi_\theta)|_{\theta=\theta_0}$$

由于数值相同，导数方向相同，所以在有限的范围内，两个目标函数的数值相差比较有限，我们就可以用近似函数替代原始目标。那么，我们沿着近似目标的导数方向做有限步长的变化，也同样可以提升策略。

既然我们知道在策略附近的区域，两个目标函数可以相互近似，就可以加入另外一个约束，用于限定新策略与旧策略的差异。如果我们把策略模型想象成一个概率分布，那么我们可以使用 KL 散度表示两个分布的距离。其中 $D_{\mathrm{KL}}^{\max}(\pi, \tilde{\pi}) = \max_s \mathrm{D}_{\mathrm{KL}}(\pi(\cdot|s)||\tilde{\pi}(\cdot|s))$，经过推导可以证明，当 $\tilde{\pi} = \mathrm{argmax}_{\pi'} L_\pi(\pi')$ 时，满足

$$\eta(\tilde{\pi}) \geqslant L_\pi(\tilde{\pi}) - \boldsymbol{C} \times D_{\mathrm{KL}}^{\max}(\pi, \tilde{\pi})$$
$$\text{where } \boldsymbol{C} = \frac{4\epsilon\gamma}{(1-\gamma)^2}$$

这部分的证明比较复杂，这里不再给出。我们令 $M_i(\pi) = L_{\pi_i}(\pi) - \boldsymbol{C} \times D_{\mathrm{KL}}^{\max}(\pi_i, \pi)$ 为第 i 轮迭代得到的值，通过推导得到

$$\eta(\pi_{i+1}) \geqslant L_{\pi_i}(\pi_{i+1}) - \boldsymbol{C} \times D_{\mathrm{KL}}^{\max}(\pi_i, \pi_{i+1}) = M_i(\pi_{i+1})$$
$$\eta(\pi_i) = L_{\pi_i}(\pi_i) = L_{\pi_i}(\pi_i) - \boldsymbol{C} \times D_{\mathrm{KL}}^{\max}(\pi_i, \pi_i) = M_i(\pi_i)$$

根据上面两个公式，可以得到

$$\eta(\pi_{i+1}) - \eta(\pi_i) \geqslant M_i(\pi_{i+1}) - M_i(\pi_i)$$

如果 $\pi_{i+1} = \mathrm{argmax}_\pi M_i(\pi)$，那么上面不等式右边就是一个非负值，这样我们就能确保策略对应的期望价值非负上升。现在目标函数得到了一定的化简，我们就可以围绕这个公式进行优化。

10.1.3 TRPO 的目标定义

下面开始介绍 TRPO 的优化算法，它的实现有一些复杂，经过前面的推导，我们已经知道算法的目标函数为

$$\text{maximize}_{\pi}[L_{\pi_{\text{old}}}(\pi) - \boldsymbol{C} \times D_{\text{KL}}^{\max}(\pi_{\text{old}}, \pi)]$$

在实践中，这个公式还有很多问题，下面我们将一一解决。

第一个优化的目标是 KL 散度。首先在实践中上式虽然可以得到比较好的效果，但是由于上式的理论验证过于保守，每一轮迭代更新的步长都偏小，导致优化的速度很慢。为了解决这个问题，我们将优化形式进行转变，得到了有约束条件的优化问题：

$$\begin{aligned}&\text{maximize}_{\pi} L_{\pi_{\text{old}}}(\pi)\\ &\text{s.t.} \quad D_{\text{KL}}^{\max}(\pi_{\text{old}}, \pi) \leqslant \delta\end{aligned}$$

当问题变成了这个形式后，原公式中较复杂的部分都被转移到了约束中。在公式中我们需要对最大值进行约束，而最大值表示为 KL 散度的上界，这实际上相当于对所有状态的 KL 散度进行约束，这样约束条件会变得多而复杂。为了简化运算，这里将最大值变成均值，虽然约束条件有所放宽，但这样可以降低计算的难度，而且从实际效果看这样不会造成效果下降。我们令

$$\overline{D}_{\text{KL}}^{\rho}(\pi_1, \pi_2) = E_{\boldsymbol{s}\sim\rho}[D_{\text{KL}}(\pi_1(\cdot|\boldsymbol{s})||\pi_2(\cdot|\boldsymbol{s}))]$$

为两个策略的平均 KL 散度，于是问题又变成了如下形式：

$$\begin{aligned}&\text{maximize}_{\pi} L_{\pi_{\text{old}}}(\pi) = \eta(\pi_{\text{old}}) + \sum_{\boldsymbol{s}} \rho_{\pi_{\text{old}}}(\boldsymbol{s}) \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}) A_{\pi_{\text{old}}}(\boldsymbol{s}, \boldsymbol{a})\\ &\text{s.t.} \quad \overline{D}_{\text{KL}}^{\rho_{\pi_{\text{old}}}}(\pi_{\text{old}}, \pi) \leqslant \delta\end{aligned}$$

第二个优化和新策略 $\pi(\boldsymbol{a}|\boldsymbol{s})$ 有关，我们可以将目标公式的第二项写作如下形式：

$$\sum_{\boldsymbol{s}} \rho_{\pi_{\text{old}}}(\boldsymbol{s}) E_{\boldsymbol{a}\sim\pi}[A_{\pi_{\text{old}}}(\boldsymbol{s}, \boldsymbol{a})]$$

如果我们想采用蒙特卡罗法对行动进行采样，就需要事先知道新策略的形式，这对优化同样造成了阻碍。为了方便计算，我们需要对其进行化简。这里采用重要性采样的方法，由于新策略不会发生特别大的变化，所以两个策略的概率分布十分接近，使用重要性采样也可以获得比较好的效果。那么公式就变为

$$\sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}) A_{\pi_{\text{old}}}(\boldsymbol{s}, \boldsymbol{a}) = \sum_{\boldsymbol{a}} \pi_{\text{old}}(\boldsymbol{a}|\boldsymbol{s}) \frac{\pi(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\text{old}}(\boldsymbol{a}|\boldsymbol{s})} A_{\pi_{\text{old}}}(\boldsymbol{s}, \boldsymbol{a})$$

$$= E_{\boldsymbol{a}\sim\pi_{\text{old}}}[\frac{\pi(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\text{old}}(\boldsymbol{a}|\boldsymbol{s})}A_{\pi_{\text{old}}}(\boldsymbol{s},\boldsymbol{a})]$$

有了这个形式，采样困难的问题就得到了解决，现在可以基于旧策略进行采样计算。2.4 节介绍过重要性采样的原理，也提到如何提高重要性采样的效果这个问题。当原始概率分布和新的概率分布十分接近时，重要性采样的效果会比较好，由于前面我们限定新旧策略的差距不会太大，所以这里的重要性采样效果不会差。

此时我们已经完成了主要的公式推导，下面就将 TRPO 目标函数转换成 3.3 节介绍的自然梯度法的形式。我们可以回顾一下，自然梯度法的目标函数为

$$\begin{aligned}&\text{minimize}_{\Delta\boldsymbol{w}} f(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} f(\boldsymbol{w})\Delta\boldsymbol{w}\\ &\text{s.t.}\quad \frac{1}{2}\Delta\boldsymbol{w}^{\mathrm{T}}\boldsymbol{I}_{f_{\boldsymbol{w}}}\Delta\boldsymbol{w} < \epsilon\end{aligned}$$

其中 $\boldsymbol{w}$ 表示参数，f 表示待优化的函数，$\boldsymbol{I}_{f_{\boldsymbol{w}}}$ 表示 Fisher 信息矩阵。对这部分内容不太熟悉的读者可以回顾 3.3 节的内容。如果我们将 TRPO 的目标函数变换成上面的形式，就能使用自然梯度法进行求解。幸运的是，我们可以用一些方法将问题转换成这样的形式。令策略 π 的参数为 θ，首先，对目标函数进行一阶泰勒展开，可以得到

$$\begin{aligned}L_{\pi_{\text{old}}}(\pi) &= L_{\pi_{\text{old}}}(\pi;\theta_{\text{old}}+\Delta\theta)\\ &\simeq L_{\pi_{\text{old}}}(\pi_{\text{old}};\theta_{\text{old}}) + \nabla_{\pi} L_{\pi_{\text{old}}}(\pi;\theta_{\text{old}})|_{\pi=\pi_{\text{old}}}(\Delta\theta)\end{aligned}$$

其次，对约束条件进行变换，首先根据 3.3 节证明得到的公式

$$\text{KL}(f(\boldsymbol{w})||f(\boldsymbol{w}+\Delta\boldsymbol{w})) = -\frac{1}{2}\Delta\boldsymbol{w}^{\mathrm{T}}E_{f_{\boldsymbol{w}}}[\nabla^2_{\boldsymbol{w}}\log f(\boldsymbol{w})]\Delta\boldsymbol{w}$$

可以对约束条件做第一步变换：

$$\begin{aligned}\overline{D}^{\rho_{\pi_{\text{old}}}}_{\text{KL}}(\pi_{\text{old}},\pi) &= \overline{D}^{\rho_{\pi_{\text{old}}}}_{\text{KL}}(\pi(\theta_{\text{old}}),\pi(\theta_{\text{old}}+\Delta\theta))\\ &= E_{\boldsymbol{s}\sim\rho}[D_{\text{KL}}(\pi(\theta_{\text{old}}|\boldsymbol{s})||\pi(\theta_{\text{old}}+\Delta\theta|\boldsymbol{s}))]\\ &= E_{\boldsymbol{s}\sim\rho}[-\frac{1}{2}\Delta\theta^{\mathrm{T}}E_{\pi_{\text{old}}}[\nabla^2_{\theta}\log\pi(\theta_{\text{old}}|\boldsymbol{s})]\Delta\theta]\end{aligned}$$

再根据 3.3 节的结论：$\boldsymbol{I}_{f_{\boldsymbol{w}}} = -E_{f_{\boldsymbol{w}}}[\nabla^2_{\boldsymbol{w}}\log f(\boldsymbol{w})]$，继续推导得到

$$= E_{\boldsymbol{s}\sim\rho}[\frac{1}{2}\Delta\theta^{\mathrm{T}}I_{\pi_{\text{old}}}(\boldsymbol{s})\Delta\theta]$$

$$\simeq \frac{1}{N}\sum_{n=1}^{N}[\frac{1}{2}\Delta\theta^{\mathrm{T}}\boldsymbol{I}_{\pi_{\mathrm{old}}}(\boldsymbol{s})\Delta\theta]$$

这样我们就从求解新策略的目标函数变成了求解策略参数更新量的目标函数：

$$\begin{aligned}&\mathrm{maximize}_{\Delta\theta}\nabla_{\pi}L_{\pi_{\mathrm{old}}}(\pi;\theta_{\mathrm{old}})|_{\pi=\pi_{\mathrm{old}}}(\Delta\theta)\\&\text{s.t.}\quad\frac{1}{N}\sum_{n=1}^{N}[\frac{1}{2}\Delta\theta^{\mathrm{T}}\boldsymbol{I}_{\pi_{\mathrm{old}}}(\boldsymbol{s})\Delta\theta]\leqslant\epsilon\end{aligned}$$

当我们完成了更新量的计算，就可以将更新量添加到原策略参数上，完成一轮迭代的优化：

$$\theta\leftarrow\theta_{\mathrm{old}}+\Delta\theta$$

经过这么长时间的推导，我们完成了 TRPO 问题的定义，然而这只是将问题定义的工作完成，后面还要继续介绍它的求解方法。在我们介绍求解之前，先来总结前面的推导思路。

- 在 10.1.1 节，我们确立了策略单调提升的概念，同时给出了两个策略差距的计算公式，我们发现如果能保证新旧策略差距项始终为正，就能确保策略模型单调非递减地更新。
- 在 10.1.2 节，为了方便计算，我们用旧策略的状态访问概率代替新策略的状态访问概率。由于只有在新旧策略相近时，两个公式才表现得相近，因此我们额外加入了衡量策略的差异 KL 散度约束。
- 本节，我们首先将无约束问题拆解成有约束问题，同时将 KL 散度的最大值改为平均值，以方便计算。接着，采用重要性采样的方法，将待采样的行动由新策略换为旧策略。最后，使用泰勒展开和 Fisher 信息矩阵的定理将公式变为可以由自然梯度法求解的目标形式。

10.1.4 自然梯度法求解

本节笔者就来介绍求解的具体过程。自然梯度法的求解方法同样采用拉格朗日乘子法，求解出最优的策略模型参数更新量为

$$\Delta\pi=\alpha\boldsymbol{I}_{\pi_{\mathrm{old}}}^{-1}\nabla_{\pi}L_{\pi_{\mathrm{old}}}(\pi;\theta_{\mathrm{old}})$$

如果直接采用这样的公式进行计算，就需要计算 Fisher 信息矩阵的逆矩阵，而逆矩阵的计算量比较大，直接计算会降低模型的训练速度，因此我们需要寻找一种方法减少这部分的计算量。对 TRPO 算法来说，它采用共轭梯度法求解更新量，从而避免了 Fisher 信息矩阵逆矩阵的计算。

我们知道共轭梯度法可以用来求解 $\boldsymbol{Ax} = \boldsymbol{g}$ 这样的问题，而参数更新量也可以表示成这样的形式：

$$I_{\pi_{\text{old}}} \Delta\pi = \nabla_{\pi} L_{\pi_{\text{old}}}(\pi; \theta_{\text{old}})$$

幸运的是，在共轭梯度法的计算过程中我们只需要矩阵 $\boldsymbol{A}$ 参与运算，这样我们就不需要计算逆矩阵了。而且在共轭梯度法的每一轮迭代中，它只参与了一次计算，就是在计算的优化步长这一步：

$$\alpha_k = \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k}$$

虽然我们节省了一定的计算时间，但是我们依然要计算出矩阵 $\boldsymbol{A}$，这个矩阵对深层神经网络这样复杂的模型来说同样是难以计算的，因此我们仍然需要找方法进行化简。幸运的是，矩阵 $\boldsymbol{A}$ 同旁边的方向向量 $\boldsymbol{p}$ 进行乘法会得到一个向量，这是简化算法的突破口。矩阵和向量乘积的结果的每一个元素都是由两个向量得到的，对应的计算公式为

$$(\boldsymbol{Ap})[i] = \sum_{j=1}^{N} (\frac{\partial}{\partial \pi_i} \frac{\partial}{\partial \pi_j} f(\pi))[i, j] \times p[j]$$

对于我们现在要解决的问题来说，求导和求和是可以互换的，所以得到

$$= \frac{\partial}{\partial \pi_i} (\sum_{j=1}^{N} \frac{\partial}{\partial \pi_j} f(\pi)[j] \times p[j])$$

这样一来，我们先进行小括号内部的计算，然后对结果进行求导，这样整体的复杂度将会大大降低。为了展示这个方法的正确性，我们举一个简单的例子。例如，我们现在有一个目标函数：

$$f(x1, x2) = x_1^2 + x_2^2 + 4x_1x_2 + 3x_1 + 4x_2$$

给定一个向量 $\boldsymbol{p}$ 为 $[\boldsymbol{p}_1, \boldsymbol{p}_2]^{\mathrm{T}}$，计算函数的矩阵 $\boldsymbol{A}$ 与向量 $\boldsymbol{p}$ 的乘积。

如果采用直接计算矩阵 $\boldsymbol{A}$ 的方法，则可以得到

$$\frac{\mathrm{d}f}{\mathrm{d}x} = [2x_1 + 4x_2 + 3, 2x_2 + 4x_1 + 4]$$

$$\boldsymbol{A}=\begin{bmatrix}2 & 4\\ 4 & 2\end{bmatrix}$$

与向量 $\boldsymbol{p}$ 相乘得到

$$\boldsymbol{A}\boldsymbol{p}=[2\boldsymbol{p}_1+4\boldsymbol{p}_2,4\boldsymbol{p}_1+2\boldsymbol{p}_2]$$

如果采用另一种方法，那么在得到一阶导数后，直接进行向量乘法，可以得到

$$\begin{aligned}\frac{\mathrm{d}f}{\mathrm{d}x}\times\boldsymbol{p}&=(2x_1+4x_2+3)\boldsymbol{p}_1+(2x_2+4x_1+4)\boldsymbol{p}_2\\&=(2\boldsymbol{p}_1+4\boldsymbol{p}_2)x_1+(4\boldsymbol{p}_1+2\boldsymbol{p}_2)x_2+3\boldsymbol{p}_1+4\boldsymbol{p}_2\end{aligned}$$

对其进行求导，得到

$$\frac{\mathrm{d}}{\mathrm{d}x}[\frac{\mathrm{d}f}{\mathrm{d}x}\boldsymbol{p}]=[2\boldsymbol{p}_1+4\boldsymbol{p}_2,4\boldsymbol{p}_1+2\boldsymbol{p}_2]$$

可见两者的结果是相同的，这也证明了这个方法的可行性。这样原本的二阶求导变成了求两次一阶导数，运算量得到了大幅降低，运算速度得到了很大的提升。解决了这个大问题，我们就可以使用共轭梯度法完成对优化方向的求解，令 $\nabla_\pi L_{\pi_{\text{old}}}(\pi;\theta_{\text{old}})$ 表示残差，令 $\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{I}_{\pi_{\text{old}}}(\boldsymbol{s})$ 为共轭梯度法的 $\boldsymbol{A}$，就可以完成对优化方向 $\boldsymbol{s}$ 的计算。有关共轭梯度法的细节，请读者回顾 3.2 节的内容。

完成了方向的计算，还需要计算在当前方向上的更新步长。由于有约束条件的存在，优化的最大步长一定存在，这样约束条件相当于为我们划定了一个置信区域（Trust Region），以保证我们的优化满足策略提升的要求。前面我们利用共轭梯度法求出了方向 $\boldsymbol{s}$，现在令最大步长为 β，也就是说，参数更新的最大值为 $\beta\boldsymbol{s}$。我们前面提到了问题的约束：

$$\frac{1}{N}\sum_{n=1}^{N}[\frac{1}{2}\Delta\theta^{\mathrm{T}}\boldsymbol{I}_{\pi_{\text{old}}}\Delta\theta]\leqslant\epsilon$$

使用 $\beta\boldsymbol{s}$ 代替 $\Delta\theta$，就可以得到

$$\frac{1}{N}\sum_{n=1}^{N}[\frac{1}{2}(\beta\boldsymbol{s})^{\mathrm{T}}\boldsymbol{I}_{\pi_{old}}(\boldsymbol{s})(\beta\boldsymbol{s})]=\frac{1}{2}\beta^2\frac{1}{N}\sum_{n=1}^{N}[\boldsymbol{s}^{\mathrm{T}}I_{\pi_{\text{old}}}(\boldsymbol{s})]\leqslant\epsilon$$

将公式进行整理，可以得到

$$\beta\leqslant\sqrt{\frac{2\epsilon}{\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{s}^{\mathrm{T}}\boldsymbol{I}_{\pi_{\text{old}}}\boldsymbol{s}}}$$

这样我们就知道了满足约束条件下的最大步长。与前面的方法类似，这里同样可以将 Fisher 信息矩阵和右边的方向向量相乘，这样可以用两个一阶导计算代替原本的二阶导计算，方法和前面介绍的类似。

最后，我们使用 backtrack 的线搜索方法找到满足优化条件的合适步长，也就是找到一个 $\pi(\theta)=\pi(\theta_{\text{old}}+\Delta\theta)$，使得它比 π_{old} 产生足够的提升。具体方法为：先尝试以 β 为步长的情况下，策略提升是否可以满足，如果已经满足则步长选择结束；如果无法满足，则将步长减少一半，再进行测试，直到满足为止。

完成了一切算法推导，我们将 TRPO 的完整算法总结出来。

算法 TRPO 算法

Input

策略网络 $\pi(\boldsymbol{a}|\boldsymbol{s};\theta)$、旧策略网络 $\pi'(\boldsymbol{a}|\boldsymbol{s};\theta')$

价值网络 $v(\boldsymbol{s};\theta)$

Start

for iter $=1$ to N :

通过交互得到交互样本 $\{\boldsymbol{s}_t,\boldsymbol{a}_t,p_t(\boldsymbol{a}|\boldsymbol{s}),\boldsymbol{r}_t,\text{done}_t\}_{i=1}^{N}$

根据实际的序列计算当前状态的目标价值 $q(\boldsymbol{s},\boldsymbol{a})$

利用 v 计算 $\boldsymbol{s}_t$ 的估计价值 $v(\boldsymbol{s}_t)$，并计算出优势值 $\text{adv}=q(\boldsymbol{s}_t,\boldsymbol{a}_t)-v(\boldsymbol{s}_t)$

根据目标函数 $\text{Loss}=\frac{1}{2}(\text{return}-v(\boldsymbol{s};\theta))^2+|\theta|_2^2$ 更新价值模型

根据上面提到的策略目标更新策略模型

10.1.5 TRPO 的实现

TRPO 的实现代码在 Baselines 项目的 baselines/trpo_mpi 中，这里我们重点关注策略模型的更新过程。有关价值模型的更新算法将在 10.2 节介绍，算法的核心代码在子项目的 trpo_mpi.py 中。从项目名称中可以看出，代码采用了 MPI 完成分布式训练，有关 MPI 的知识读者可以回顾 4.3 节的内容。这里我们只关注策略模型的目标函数构建与优化过程。

在模型方面，代码构建了两个策略模型 pi 和 oldpi，分别代表待优化的策略和当前的策略。每一次优化时我们都会固定 oldpi 优化 pi。策略更新量的计算可以分为下面几个步骤。

（1）计算目标函数的梯度：$\nabla_\pi L_{\pi_{\text{old}}}(\pi;\theta_{\text{old}})$。

（2）准备好矩阵 $\boldsymbol{A}$。

（3）使用共轭梯度法求解更新方向。

（4）求出步长的上限 β。

（5）求出真实的步长。

下面就来看看这 5 个步骤的计算过程。首先是第 1 步，目标函数的计算图结构如图 10-1 所示，其中方框框住的变量为 placeholder。

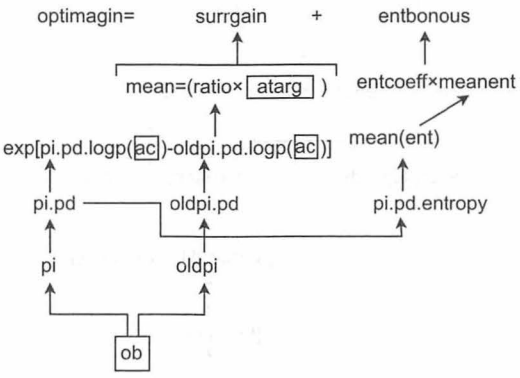


图 10-1 目标函数的计算图

图 10-1 所示为目标公式的计算过程，首先目标由 TRPO 算法中的目标函数 surrgain 和新策略的熵 entbonous 组成。熵的计算比较简单，它由新策略 pi 的熵和系数 entcoeff 组成。我们将 surrgain 继续拆解成新旧策略的比例 ratio 和优势函数 atarg 两部分，其中 atarg 会在前一步算好并传入其中。ratio 又可以继续拆解成两个策略的对数似然，这样只要计算对应行动 ac 的对数似然即可。

第 2 步需要计算矩阵 $\boldsymbol{A}$，我们可以先求出 KL 散度的一阶导，计算流程图如图 10-2 所示。由于是 KL 散度的均值，因此计算过程比较简单。为了后续计算的方便，我们可以先把矩阵的一阶导计算好，得到 klgrads。

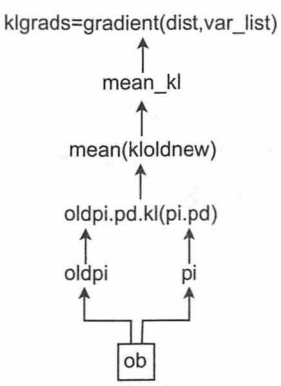


图 10-2 矩阵 $\boldsymbol{A}$ 的一阶导计算图

得到了 KL 散度的一阶导后，我们就可以构建一阶导和优化方向 $\boldsymbol{p}$ 的乘积。这个函数的计算图如图 10-3 所示。由于我们需要让矩阵 $\boldsymbol{A}$ 的一阶导和向量 $\boldsymbol{p}$ 进行点乘，因此需要把向量 $\boldsymbol{p}$ 变换成需要的形式。同时，方法 U.flatgrad 表示求出导数的同时将参数转换成一维向量的形式，这样可以继续在共轭梯度法中计算。

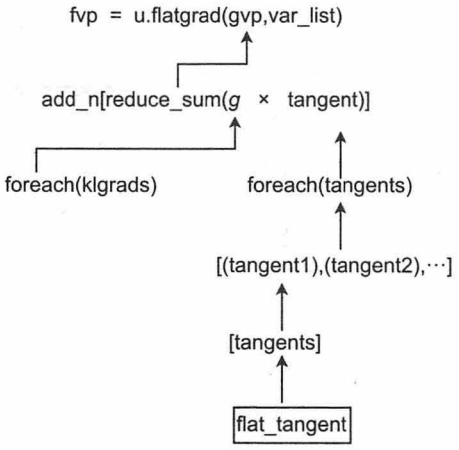


图 10-3　矩阵 $\boldsymbol{A}$ 和向量 $\boldsymbol{p}$ 的乘积的计算图

第 3 步就是使用共轭梯度法，这部分计算在计算图定义之外。现在我们已经有了梯度和 $\boldsymbol{Ap}$ 的计算方法，直接调用共轭梯度法就可以计算优化方法。第 4 步和第 5 步的计算也在实际优化中，这两部分的计算比较简单，在此不再赘述。

以上就是 TRPO 算法实现的全过程，由于这部分的内容比较复杂，希望读者能够仔细阅读代码，了解更多的细节。

10.2　GAE

在 10.1 节介绍的 TRPO 算法中，我们主要关注了策略网络的改进方法，并没有介绍价值网络的改进方法。本节介绍这部分的改进方案泛化优势估计（Generalized Advantage Estimation，简称 GAE）的具体实现。

10.2.1　GAE 的公式定义

我们已经知道在策略梯度法中，如果直接使用 On-Policy 的方法交互采样，并用每一时刻的回报作为梯度中的长期回报估计 $\sum_{t'=t}^{T}\gamma^{t'-t}\boldsymbol{r}_{t'}$，会使算法产生较大的波动，

换句话说，梯度的方差会比较大。如果采用 Actor-Critic 的方法，通过模型估计状态的价值，那么模型优化的方差会减小，但是由于函数拟合的问题，这个方法会产生一定偏差。因此问题的关键就在于如何平衡偏差和方差带来的影响。

Actor Critic 的价值梯度可以表示为

$$\nabla_\theta J(\theta) = E_{\boldsymbol{s},\boldsymbol{a}\sim\boldsymbol{\tau}}[\sum_{t=0}^{\infty} A^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)]$$

其中

$$A^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t) = Q^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t) - V^{\pi,\gamma}(\boldsymbol{s}_t)$$

$$Q^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t) = E_{\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1}\sim\boldsymbol{\tau}}[\sum_{l=0}^{\infty}\gamma^l \boldsymbol{r}_{t+l}]$$

$$V^{\pi,\gamma}(\boldsymbol{s}_t) = E_{\boldsymbol{s}_{t+1},\boldsymbol{a}_t\sim\boldsymbol{\tau}}[\sum_{l=0}^{\infty}\gamma^l \boldsymbol{r}_{t+l}]$$

总的来说，$A^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t)$ 已经可以做到在保持无偏差的情况下，尽可能地降低方差值。如果我们能通过学习得到一个完美的优势函数，模型就可以得到很好的表现。但实际中直接学习优势函数比较困难，我们往往需要组合其他函数得到优势函数，同时还需要考虑偏差和方差对模型的影响，为此我们给出了一个定义：γ-just。当一个函数 $\hat{A}_t$ 满足 γ-just 条件时，它就满足下面的公式：

$$E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\boldsymbol{\tau}}[\hat{A}_t(\boldsymbol{s}_{0:\infty},\boldsymbol{a}_{0:\infty})\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)] = E_{\boldsymbol{s}_0,\boldsymbol{a}_0,\ldots\sim\boldsymbol{\tau}}[A^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)]$$

如果我们找到的估计函数能够满足上面的公式，它就可以用来替换优势函数。经过推导分析我们发现，$\boldsymbol{r}_t + \gamma V^{\pi,\gamma}(\boldsymbol{s}_{t+1}) - V^{\pi,\gamma}(\boldsymbol{s}_t)$ 满足上述条件，是一个合格的替换项，于是后面的工作将围绕它替换进行。

令 V 为一个近似的值函数，我们定义 $\boldsymbol{\delta}_t^V = \boldsymbol{r}_t + \gamma V(\boldsymbol{s}_{t+1}) - V(\boldsymbol{s}_t)$，这里的 $\boldsymbol{\delta}_t^V$ 可以作为 $\boldsymbol{a}_t$ 的一个优势价值估计。如果上面的公式中 $V = V^{\pi,\gamma}$，那么 $\boldsymbol{\delta}_t^V$ 就是一个 γ-just 的估计函数，它可以得到 $A^{\pi,\gamma}$ 的一个无偏估计：

$$\begin{aligned} E_{\boldsymbol{s}_{t+1}}[\boldsymbol{\delta}_t^{V^{\pi,\gamma}}] &= E_{\boldsymbol{s}_{t+1}}[\boldsymbol{r}_t + \gamma V^{\pi,\gamma}(\boldsymbol{s}_{t+1}) - V^{\pi,\gamma}(\boldsymbol{s}_t)] \\ &= E_{\boldsymbol{s}_{t+1}}[Q^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t) - V^{\pi,\gamma}(\boldsymbol{s}_t)] = A^{\pi,\gamma}(\boldsymbol{s}_t,\boldsymbol{a}_t) \end{aligned}$$

接下来我们考虑 n 步的优势函数估计，并用 $\hat{A}_t^{(k)}$ 表示，可以得到

$$
\begin{aligned}
\hat{A}_t^{(1)} &= \boldsymbol{\delta}_t^V = -V(\boldsymbol{s}_t) + \boldsymbol{r}_t + \gamma V(\boldsymbol{s}_{t+1}) \\
\hat{A}_t^{(2)} &= \boldsymbol{\delta}_t^V + \gamma \boldsymbol{\delta}_{t+1}^V \\
&= -V(\boldsymbol{s}_t) + \boldsymbol{r}_t + \gamma V(\boldsymbol{s}_{t+1}) + \gamma(-V(\boldsymbol{s}_{t+1}) + \boldsymbol{r}_{t+1} + \gamma V(\boldsymbol{s}_{t+2})) \\
&= -V(\boldsymbol{s}_t) + \boldsymbol{r}_t + \gamma \boldsymbol{r}_{t+1} + \gamma^2 V(\boldsymbol{s}_{t+2})
\end{aligned}
$$

依此类推，可以得到

$$
\hat{A}_t^{(\infty)} = \sum_{l=0}^{\infty} \gamma^l \boldsymbol{\delta}_{t+l}^V = -V(\boldsymbol{s}_t) + \boldsymbol{r}_t + \gamma \boldsymbol{r}_{t+1} + \cdots + \gamma^k \boldsymbol{r}_{t+k} + \cdots
$$

我们知道 γ 是一个小于 1 的数，随着 k 趋近于无穷大，最终 $\gamma^{\infty} V(\boldsymbol{s}_{t+\infty}) \to 0$，所以 $\hat{A}_t^{(\infty)}$ 这一项相当于用蒙特卡罗法对优势函数进行估计。

此时我们看到，随着估计步数的增加，估计值的偏差逐渐变小，方差逐渐变大。如果我们能将这些估计值同时考虑在内，是不是就可以在偏差和方差之间找到更好的平衡呢？基于这样的想法，我们可以得到多个估计值加权平均后的公式，公式的计算过程如图 10-4 所示。

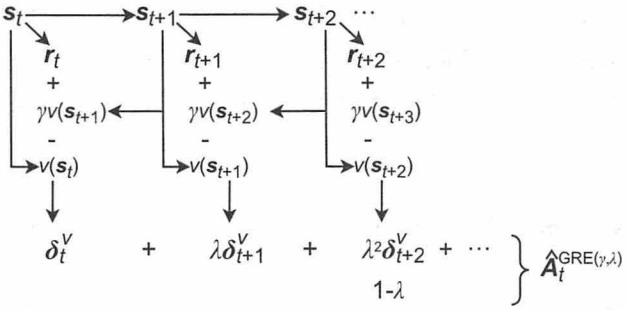


图 10-4　GAE 计算公式图解

$$
\begin{aligned}
\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} &= (1-\lambda)(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \cdots) \\
&= (1-\lambda)(\boldsymbol{\delta}_t^V + \lambda(\boldsymbol{\delta}_t^V + \boldsymbol{\delta}_{t+1}^V) + \lambda^2(\boldsymbol{\delta}_t^V + \gamma \boldsymbol{\delta}_{t+1}^V + \gamma^2 \boldsymbol{\delta}_{t+2}^V) + \cdots) \\
&= (1-\lambda)(\boldsymbol{\delta}_t^V(1 + \lambda + \lambda^2 + \cdots) + \gamma \boldsymbol{\delta}_{t+1}^V(\lambda + \lambda^2 + \lambda^3 + \cdots) \\
&\quad + \gamma^2 \boldsymbol{\delta}_{t+2}^V(\lambda^2 + \lambda^3 + \lambda^4 + \cdots) + \cdots) \\
&= (1-\lambda)(\boldsymbol{\delta}_t^V(\frac{1}{1-\lambda}) + \gamma \boldsymbol{\delta}_{t+1}^V(\frac{\lambda}{1-\lambda}) + \gamma^2 \boldsymbol{\delta}_{t+2}^V(\frac{\lambda^2}{1-\lambda}) + \cdots) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \boldsymbol{\delta}_{t+l}^V
\end{aligned}
$$

我们发现这个公式的最终形式比较简洁，虽然我们引入了一个新的超参数，但是公式并没有复杂太多。此时我们的估计值在偏差和方差方面得到了更好的平衡，我们可以分别计算 λ 等于 0 和 1 时的值

$$\begin{aligned}
&\text{GAE}(\gamma,0): \quad \hat{A}_t := \boldsymbol{\delta}_t \quad\quad = \boldsymbol{r}_t + \gamma v(\boldsymbol{s}_{t+1}) - v(\boldsymbol{s}_t) \\
&\text{GAE}(\gamma,1): \quad \hat{A}_t := \textstyle\sum_{l=0}^{\infty} \gamma^l \boldsymbol{\delta}_{t+l} \quad = \text{sum}_{t=0}^{\infty} \gamma^l \boldsymbol{r}_{t+l} - v(\boldsymbol{s}_t)
\end{aligned}$$

可以看出，当 $\lambda = 0$ 时，算法等同于计算 TD-Error，这是一个方差较低但偏差较高的算法；当 $\lambda = 1$ 时，算法变成蒙特卡罗目标值和价值估计的差，这是一个偏差较低但方差较高的算法。我们可以通过调整 λ 值使模型在两者之间得到更好的平衡。因此，我们可以用它代替前面公式中的优势函数，此时计算的公式就变为

$$\nabla_\theta J(\theta) = E_{\boldsymbol{s},\boldsymbol{a}\sim\boldsymbol{\tau}}[\sum_{t=0}^{\infty} \hat{A}_t^{\text{GAE}(\gamma,\lambda)} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)]$$

10.2.2 基于 GAE 和 TRPO 的值函数优化

在介绍 GAE 的论文 *High-Dimensional Continuous Control Using Generalized Advantage Estimation*[2] 中，算法除了将 TRPO 方法应用在策略的训练上，还将 TRPO 应用到了价值网络的训练上。价值网络的目标函数为

$$\text{minimize}_\phi \sum_{n=1}^{N} ||V_\phi(\boldsymbol{s}_n) - \hat{V}_n||^2$$

我们可以使用类似 Trust Region 的方法对目标值函数进行限定。限定的核心思想是希望新的价值模型与旧模型不要相差过大，那么我们首先可以得到原始值函数和 GAE 方法的估计值的均方误差 σ^2：

$$\sigma^2 = \frac{1}{N} \sum_{n=1}^{N} ||V_{\phi_{\text{old}}}(\boldsymbol{s}_n) - \hat{V}_n||^2$$

我们希望新旧模型的价值估计差不要过大，为了使约束具有一定的自适应性，我们将限定值设定为和 σ^2 相关的一个值，于是我们的问题变为

$$\begin{aligned}
&\text{minimize}_\phi \sum_{n=1}^{N} ||V_\phi(\boldsymbol{s}_n) - \hat{V}_n||^2 \\
&\text{s.t.} \quad \frac{1}{N} \sum_{n=1}^{N} \frac{||V_\phi(\boldsymbol{s}_n) - V_{\phi_{\text{old}}}(\boldsymbol{s}_n)||^2}{2\sigma^2} \leqslant \epsilon
\end{aligned}$$

这个问题的描述和 TRPO 中对策略网络的描述很类似，约束条件也和 TRPO 中的 KL 散度约束类似，这样我们同样可以把问题转变为一阶目标函数和二阶约束条件的形式：

$$\begin{aligned}&\text{minimize}_{\phi}\nabla_{\phi}V_{\phi}(\boldsymbol{s}_n)^{\mathrm{T}}(\phi-\phi_{\text{old}})\\&\text{s.t.}\quad\frac{1}{N}\sum_{n=1}^{N}(\phi-\phi_{\text{old}})^{\mathrm{T}}H(\phi-\phi_{\text{old}})\leqslant\epsilon\end{aligned}$$

其中 $H=\frac{1}{N}\sum_n j_n j_n^{\mathrm{T}}$，$j_n=\nabla_{\phi}V_{\phi}(\boldsymbol{s}_n)$。得到了这样的形式，我们就可以用 Fisher 信息矩阵近似值函数的二阶导，同时利用共轭梯度法进行优化，通过计算矩阵和向量的乘积来减少计算。由于这部分的计算过程和 TRPO 类似，这里就不再赘述。通过这种估计方法，我们在使用 TRPO 方法估计时可以获得更稳定的模型，效果也比原始模型更好。

10.2.3 GAE 的实现

在 Baselines 项目的 trpo_mpi 目录中，我们可以看到 GAE 算法的应用，算法的实现在 trpo_mpi.py 文件的 add_vtarg_and_adv 方法上。通过与环境交互，我们得到了每一时刻模型对状态 $\boldsymbol{s}$ 的价值估计值 $v(\boldsymbol{s})$，这样就可以计算出每一时刻的 TD Error：$\boldsymbol{\delta}_t=\boldsymbol{r}_t+\gamma v(\boldsymbol{s}_{t+1})-v(\boldsymbol{s}_t)$，然后应用 GAE 的公式 $\sum_{l=0}^{\infty}(\gamma\lambda)^l\boldsymbol{\delta}_{t+l}^V$ 求解出优势函数的估计值。为了快速求出序列中所有时刻的估计值，代码采用了倒序的计算方式，通过 $t+1$ 时刻的优势估计值辅助计算 t 时刻的估计值。我们可以通过公式展开得到

$$\begin{aligned}\hat{A}_t^{\text{GAE}(\gamma,\lambda)}&=\sum_{l=0}^{\infty}(\gamma\lambda)^l\boldsymbol{\delta}_{t+l}^V\\&=\boldsymbol{\delta}_t^V+\sum_{l=1}^{\infty}(\gamma\lambda)^l\boldsymbol{\delta}_{t+l}^V\\&=\boldsymbol{\delta}_t^V+\gamma\lambda\sum_{l=0}^{\infty}(\gamma\lambda)^l\boldsymbol{\delta}_{t+1+l}^V\\&=\boldsymbol{\delta}_t^V+\gamma\lambda\hat{A}_{t+1}^{\text{GAE}(\gamma,\lambda)}\end{aligned}$$

这样就完成了基于 GAE 的目标价值计算，虽然整个过程有些复杂，但是我们获得了更大的灵活度，我们可以根据具体问题找到更适合的价值计算方法。

10.3 PPO

10.3.1 PPO 介绍

PPO 是一个基于 TRPO 算法的改进算法，前面我们已经知道，TRPO 算法的问题定义如下所示：

$$\begin{aligned}&\text{maximize}\ \ E_{\boldsymbol{a}\sim\pi_{\text{old}}}[\frac{\pi(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\text{old}}(\boldsymbol{a}|\boldsymbol{s})}A_{\pi_{\text{old}}}(\boldsymbol{s},\boldsymbol{a})]\\&\text{s.t.}\ \ \ \overline{D}_{\text{KL}}^{\rho_{\pi_{\text{old}}}}(\pi_{\text{old}},\pi)\leqslant\delta\end{aligned}$$

TRPO 算法使用共轭梯度法进行求解更新，这样可以减少 Fisher 信息矩阵的计算量，但是实际上它的计算量仍然比较大，算法的速度和其他基于策略梯度的算法相比仍有一定的差距。同时，共轭梯度法和梯度下降法相比略显复杂，在实现上也更困难，因此论文的作者希望将目标函数进行一定的变换，并使用常见的梯度下降法进行求解，这样可以在实现难度和求解效率上得到提升。

由于实际中我们希望采用蒙特卡罗法完成对期望的近似，因此目标 TRPO 的函数也可以变为

$$\text{maximize}\ \ \frac{1}{N}\sum_{t=1}^{N}[\frac{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}{\pi_{\text{old}}(\boldsymbol{a}_t|\boldsymbol{s}_t)}A_{\pi_{\text{old}}}(\boldsymbol{s}_t,\boldsymbol{a}_t)]$$

此时我们用一个变量 $r_t(\theta)$ 表示公式中的 $\frac{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}{\pi_{\text{old}}(\boldsymbol{a}_t|\boldsymbol{s}_t)}$，它代表了两个策略概率的比率。这样上面的目标函数就变为

$$\text{maximize}\ \ \frac{1}{N}\sum_{t=1}^{N}[r_t(\theta)A_{\pi_{\text{old}}}(\boldsymbol{s}_t,\boldsymbol{a}_t)]$$

这样的变换有什么意义呢？从 TRPO 的约束条件中可以看出，算法对模型每一轮的优化都划定了置信区域，为了保证优化的稳定性，每一步优化都不能超过这个范围，而范围的限定就是由约束条件中的 KL 散度给出的。直观地看，约束条件要求新的策略概率分布不能与旧的概率分布差距太大，于是我们可以认为，TRPO 的约束条件和要求 $r_t(\theta)$ 的值靠近 1 是相似的，于是作者提出了另一种目标函数：

$$L_t^{\text{CLIP}}=\frac{1}{N}\sum_{t=1}^{N}[\min(r_t(\theta)A_{\pi_{\text{old}}}(\boldsymbol{s}_t,\boldsymbol{a}_t),\text{clip}(r_t(\theta),1-\epsilon,1+\epsilon)A_{\pi_{\text{old}}}(\boldsymbol{s}_t,\boldsymbol{a}_t))]$$

书籍是人类进步的阶梯

可以看出目标函数为了确保算法能达到 TRPO 中置信区域的效果，为算法上了两把“锁”：第一把锁是对新旧策略概率比率 $r_t(\theta)$ 的限制，这个比率只能限制在 $[1-\epsilon, 1+\epsilon]$，这确保了每一次的更新不会有太大的波动；第二把锁就是 min 函数，它将在两个结果中选择一个较低值作为结果。如果我们能够将较低值优化到令人满意的程度，那么对于其他的情况，模型一定会表现得更好。

除了上面的 L_t^{CLIP} 目标，作者又给出了另一种目标函数。在这个目标函数中，公式将采用另一个变量控制约束项和目标项之间的权重关系，这个方法被称为 Adaptive KL Penalty 方法，其中新加入的参数 β 被称为 Adaptive KL Penalty Coefficient。新的目标函数就变为

$$L_t^{\text{KLPEN}}(\theta) = \frac{1}{N}\sum_{t=1}^{N}[\frac{\pi(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\text{old}}(\boldsymbol{a}|\boldsymbol{s})}A_{\pi_{\text{old}}}(\boldsymbol{s},\boldsymbol{a}) - \beta\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|\boldsymbol{s}_t), \pi_\theta(\cdot|\boldsymbol{s}_t)]]$$

那么 β 该如何确定呢？我们可以使用两个分布的 KL 散度值进行调节。这里设定一个目标值 d_{targ} 为计算训练数据平均 KL 散度的阈值，我们可以计算出 KL 散度的平均值：

$$d = \frac{1}{N}\sum_{t=1}^{N}[\text{KL}(\pi_{\theta_{\text{old}}}(\cdot|\boldsymbol{s}_t), \pi_\theta(\cdot|\boldsymbol{s}_t))]$$

接下来就要根据 d 和 d_{targ} 的关系调整 β：

- 如果 $d < d_{\text{targ}}, \beta \leftarrow \beta/2$，相当于放松 KL 约束限制。
- 如果 $d > d_{\text{targ}}, \beta \leftarrow \beta \times 2$，相当于增强 KL 约束限制。

更新后的 β 值将在下一轮优化中发挥它的作用。

这就是论文作者提出的两种目标函数。经过试验对比，作者最终选择了 L_t^{CLIP} 作为 TRPO 目标函数的替代。在实际实现过程中，PPO 方法还可以结合其他的方法对目标公式中的优势函数做近似，例如在 9.2 节介绍的 A3C 算法中实用的 N 步回报价值估计法：

$$\hat{A}_t = -V(\boldsymbol{s}_t) + \boldsymbol{r}_t + \gamma\boldsymbol{r}_{t+1} + \cdots + \gamma^{T-t+1}\boldsymbol{r}_{T-1} + \gamma^{T-t}V(\boldsymbol{s}_T)$$

以及 10.2 节介绍的 GAE 方法：

$$\begin{aligned}\hat{A}_t &= \boldsymbol{\delta}_t + (\gamma\lambda)\boldsymbol{\delta}_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\boldsymbol{\delta}_{T-1}\\ \boldsymbol{\delta}_t &= \boldsymbol{r}_t + \gamma V(\boldsymbol{s}_{t+1}) - V(\boldsymbol{s}_t)\end{aligned}$$

由于这些方法已经在前面介绍过，这里不再赘述。除了关于策略的目标函数，PPO

书籍是人类进步的阶梯

方法还将值函数的目标和策略模型的熵加入目标，于是完整的目标函数变为

$$L_t^{\text{CLIP+VF+}S}(\theta) = \hat{E}_t[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](\boldsymbol{s}_t)]$$

其中 $L_t^{\text{VF}} = (V_\theta(\boldsymbol{s}_t) - V_t^{\text{target}})^2$

由于 PPO 算法采用常见的梯度下降法求解，因此求解过程比较简单，这里不再赘述。

10.3.2 PPO 算法实践

Baselines 项目包含了两个 PPO 算法的项目：baselines/ppo1 和 baselines/ppo2，接下来介绍 ppo1 中的内容。这里代码的结构和 TRPO 中的代码结构类似，其中的 cnn_policy.py 和 mlp_policy.py 中包含了两种策略模型。模型的结构和 A2C 模型的结构类似：

- 输入为状态 $\boldsymbol{s}$，策略模型和价值模型共享部分模型。
- 策略部分的输出为行动的概率分布参数 θ，我们可以根据分布 $\pi(\boldsymbol{a}|\boldsymbol{s};\theta)$ 采样得到行动。
- 价值部分的输出为当前状态的价值 $v(\boldsymbol{s})$。

核心的算法实现在 pposgd_simple.py 中，算法同样使用 MPI 进行分布式训练，在构建计算图的部分，代码构建了类似于 TRPO 的目标函数，构建过程如图 10-5 所示。

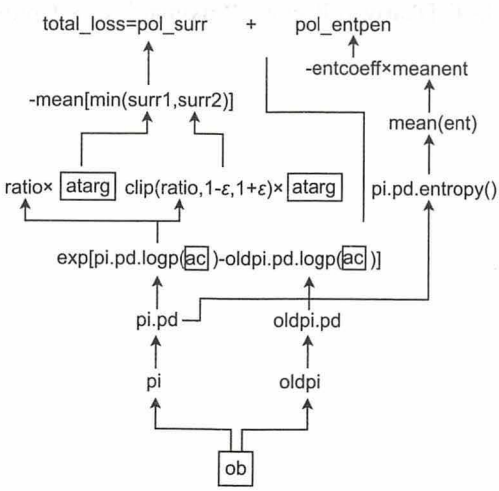


图 10-5 PPO 的计算图

书籍是人类进步的阶梯

实际上 PPO 的计算图和 TRPO 的类似，在 10.1 节中已经详细分析过，这里不再赘述，项目的其他代码请读者自行阅读。

10.4 总结

本章介绍了使策略单调提升的优化方法，下面让我们来总结一下。

（1）为了确保策略能够单调提升，我们需要找到比较策略间差异的公式。

（2）TRPO 算法构建了类似自然梯度法的目标公式，并使用共轭梯度法减少 Fisher 信息矩阵的计算量。

（3）GAE 算法通过融合多种 TD-Error 使目标价值的计算可以在偏差和方差之间做到更好的平衡。

（4）PPO 算法通过化简目标函数并使用梯度下降法加快了学习速度。

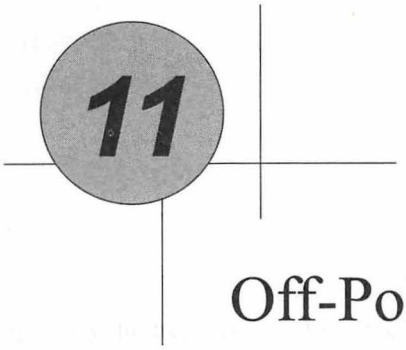
10.5 参考资料

[1] Schulman J, Levine S, Moritz P, et al. Trust Region Policy Optimization[C]// ICML. 2015.

[2] Schulman J, Moritz P, Levine S, et al. High-Dimensional Continuous Control Using Generalized Advantage Estimation[J]. Computer Science, 2015.

[3] Schulman J, Wolski F, Dhariwal P, et al. Proximal Policy Optimization Algorithms[J]. 2017.

书籍是人类进步的阶梯



Off-Policy 策略梯度法

我们已经介绍了很多有关策略梯度的算法，这些算法也取得了不错的成绩，但是它们仍然要面对策略梯度的两个软肋。这两个软肋已经在前面提到，笔者总结如下。

- **高方差**：由于每一次交互我们都能得到一条轨迹，基于这条轨迹的信息进行策略更新时会造成回报估计的波动，影响最终效果。为了减少方差对算法的影响，Actor Critic 方法增加了一个模型，用于估计当前状态的价值，通过引入一定的偏差换取方差的降低。
- **样本使用率**：这个问题是所有 On-Policy 算法都要解决的。前面提到策略梯度针对当前策略的价值希望进行优化，所以需要使用 On-Policy 的方法进行优化，这带来了两个问题。首先，On-Policy 方法并不是一个节约样本的方法，每一次策略发生改变，都要丢弃前面产生的样本，这将带来很大的样本浪费。对于通过计算机模拟进行的任务来说，这点浪费似乎不算什么，但是对于物理环境中的任务来说，机器人与真实环境的交互是十分耗时的，每次丢掉大量样本并不是一个好主意，所以我们需要考虑用 Off-Policy 的算法进行学习。

第 10 章介绍了用 TRPO 和 PPO 方法解决第一个问题，减少算法的波动，提高了算法的稳定性，甚至为策略模型提供了单调上升的性质；而本章我们要解决第二个问题，即使用 Off-Policy 的方法。第 7 章和第 8 章介绍了以 DQN 为核心思想的基于价值模型的 Off-Policy 算法，以及包括 Replay Buffer 在内的一系列结构，本章还会见到这些熟悉的身影。

书籍是人类进步的阶梯

11.1 Retrace

从本节开始，我们要使用由其他策略交互得到的样本对当前的模型进行训练，由于样本并非来自同一个样本，两者的概率分布也并不相同，因此我们无法直接使用这些样本。2.4 节介绍了“使用概率分布 A 为概率分布 B 采样”的方法——重要性采样，我们可以使用这样的方法解决概率分布不匹配的问题。本节，我们先来介绍如何估计 Off-Policy 的轨迹价值，并介绍 Retrace 算法。

11.1.1 Retrace 的基本概念

基于 Off-Policy 的价值估计方法主要使用重要性采样的方法实现，我们可以用一个 R 表示这一类计算方法的基本形式：

$$\mathrm{R}Q_\pi(\boldsymbol{x},\boldsymbol{a}) = Q_\pi(\boldsymbol{x},\boldsymbol{a}) + E_\mu[\sum_{t\geqslant 0}\gamma^t(\prod_{s=1}^{t} c_s)(\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t))]$$

其中 $Q(\boldsymbol{x},\boldsymbol{a})$ 表示值函数估计值，μ 表示参与交互的策略，π 表示待学习的策略，γ 表示回报的打折率：

$$c_s = \frac{\pi(\boldsymbol{a}_s|\boldsymbol{x}_s)}{\mu(\boldsymbol{a}_s|\boldsymbol{x}_s)}$$

表示新旧策略的概率比率。

我们先考虑 μ 和 π 完全相同的情况，此时公式中的 $\prod_{s=1}^{t} c_s$ 等于 1。当 $t=0$ 时，上面的公式就变成了 Actor Critic 中 TD-Error 的计算公式：

$$\mathrm{R}Q_\pi(\boldsymbol{x},\boldsymbol{a}) = \boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot)$$

如果时间长度进一步拉长，我们可以得到

$$\begin{aligned}\mathrm{R}_{t=1}Q_\pi(\boldsymbol{x},\boldsymbol{a}) &= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + E_\pi[\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1}|\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t) + \gamma(\boldsymbol{r}_{t+1} + \gamma Q_\pi(\boldsymbol{x}_{t+2}|\cdot) \\ &\quad - Q_\mu(\boldsymbol{x}_{t+1},\boldsymbol{a}_{t+1}))] \\ &= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + E_\pi[\sum_{d=0}^{1}\gamma^d(\boldsymbol{r}_{t+d} + \gamma Q_\pi(\boldsymbol{x}_{t+d+1}|\cdot) - Q_\mu(\boldsymbol{x}_{t+d},\boldsymbol{a}_{t+d}))\end{aligned}$$

此时的公式形式和 10.2 节中 GAE 的计算公式比较接近。

书籍是人类进步的阶梯

当两个策略变得不同时，$\prod_{s=1}^{t} c_s$ 这个项目就显得比较重要了。将上面的公式展开就可以得到

$$
\begin{aligned}
&\mathrm{R}Q_\pi(\boldsymbol{x},\boldsymbol{a}) \\
&= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + E_\pi[\sum_{t\geqslant 0}\gamma^t(\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t))] \\
&= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + \sum_{\boldsymbol{\tau}}[\prod_t \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)\sum_{t\geqslant 0}\gamma^t(\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t))] \\
&= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + \sum_{\boldsymbol{\tau}}[\prod_t \mu(\boldsymbol{a}_t|\boldsymbol{s}_t)\frac{\prod_t \pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}{\prod_t \mu(\boldsymbol{a}_t|\boldsymbol{s}_t)}\sum_{t\geqslant 0}\gamma^t(\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t))] \\
&= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + \sum_{\boldsymbol{\tau}}[\prod_t \mu(\boldsymbol{a}_t|\boldsymbol{s}_t)\sum_{t\geqslant 0}\gamma^t\prod_t\frac{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}{\mu(\boldsymbol{a}_t|\boldsymbol{s}_t)}(\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t))] \\
&= Q_\pi(\boldsymbol{x},\boldsymbol{a}) + E_\mu[\sum_{t\geqslant 0}\gamma^t\prod_t\frac{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}{\mu(\boldsymbol{a}_t|\boldsymbol{s}_t)}(\boldsymbol{r}_t + \gamma Q_\pi(\boldsymbol{x}_{t+1},\cdot) - Q_\mu(\boldsymbol{x}_t,\boldsymbol{a}_t))]
\end{aligned}
$$

重要性采样算法本身是无偏的，但是当两个分布相差比较大时，重要性采样算法的效果就会很差，而且在数值上可能出现一些问题。当 $\mu(\boldsymbol{a}_s|\boldsymbol{x}_s)$ 非常小而 $\pi(\boldsymbol{a}_s|\boldsymbol{x}_s)$ 非常大时，这个比值就会变得非常大，从而使模型产生较大的波动。所以直接使用重要性采样进行计算不能保证算法的稳定性，我们需要对这个比率做更多的限制。

为了解决这个问题，Retrace(λ) 算法（在本节中假设 $\lambda = 1$）采用了对比率进行限定的方式，使算法更新变得更稳定：

$$c_t = \min(1, \frac{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)}{\mu(\boldsymbol{a}_t|\boldsymbol{s}_t)})$$

通过这样的设定，比率既存在一个上限，不至于因为比率差距过大而对模型产生过大的波动，又可以保持重要性采样的基本形式，使计算不会有太大的偏差。我们可以证明得到 Retrace 算法同样具有良好的收敛性质，并使用它进行价值更新。更多的证明过程请读者阅读论文 *Safe and efficient off-policy reinforcement learning*[1] 了解更多的细节。

11.1.2 Retrace 的算法实现

Retrace 算法该如何实现呢？在 Baselines 项目的 baselines/acer/acer_simple.py 文件中可以看到它的实现。这个子项目实现了 ACER 算法的全过程，其中包含了 Retrace 的实现。在 Retrace 的代码中，我们可以看到如下参数介绍：

```
def q_retrace(R, D, q_i, v, rho_i, nenvs, nsteps, gamma):
    """
    nenvs:表示参与计算的episode数量，由于算法涉及多进程，所以实际上有nenvs
        个进程进行交互采样，每个进程完成1个episode的采样，这样就有nenvs个
        episode
    nsteps:表示每个episode的长度
    R:记录了每一时刻的Reward，维度为(nenvs, nsteps)
    D:记录了每一时刻episode是否已经完成，维度为(nenvs, nsteps)
    q_i:由模型计算得到的每一时刻的Q(s,a)
    v:由模型计算得到的每一时刻的V(s)，计算方法为sum([p_i[i] * q_i[i] for i
        in actions])，p_i表示某个行动的发生概率
    rho_i:两个策略的比率

    函数返回：经过Q_retrace计算得到的Return值
    """
```

代码中已经假设 Retrace 算法中的 λ 值为 1，代码中 Retrace($\lambda = 1$) 算法和论文中提到的公式有一些不同，我们需要重新做推导来说明实现过程采用的公式。首先，我们使用 Expected SARSA 的价值更新公式：

$$Q_\pi^{\text{ret}}(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow E_{\boldsymbol{a}\sim\pi}[\boldsymbol{r}_t + \gamma Q_\pi^{\text{ret}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1})]$$

SARSA 是一个 On-Policy 的算法，我们可以通过重要性采样的方法将其变成一个 Off-Policy 的方法：

$$Q_\pi^{\text{ret}}(\boldsymbol{s}_t, \boldsymbol{a}_t) = E_{\boldsymbol{a}\sim\mu}[\frac{\pi(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1})}{\mu(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1})}(\boldsymbol{r}_t + \gamma Q_\pi^{\text{ret}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}))]$$

用 ρ_{t+1} 代替 $\frac{\pi(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1})}{\mu(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1})}$，并将原本的值函数包含在公式中，可以得到

$$\begin{aligned} Q_\pi^{\text{ret}}(\boldsymbol{s}_t, \boldsymbol{a}_t) &= E_{\boldsymbol{a}\sim\mu}[\rho_{t+1}(\boldsymbol{r}_t + \gamma Q_\pi^{\text{ret}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}))] + V_\pi(\boldsymbol{s}_t) - V_\pi(\boldsymbol{s}_t) \\ &= E_{\boldsymbol{a}\sim\mu}[\rho_{t+1}(\boldsymbol{r}_t + \gamma Q^{\text{ret}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) - Q_\mu(\boldsymbol{s}_t))] + V_\pi(\boldsymbol{s}_t) \end{aligned}$$

在计算值函数时我们就可以使用这个公式进行更新。下面就来考虑具体实现中的一些问题。现实中，我们需要对模型实际的轨迹长度进行限制，即使真实的轨迹长度超过了我们的限定，我们也会将其提前终止，所以对于每一个状态，我们必须使用额外的变量 d_t 表示时刻 t 的状态是否为终止状态。其中 $d_t = 1$ 表示该轨迹终止，$d_t = 0$ 表示

该轨迹仍在继续，那么我们就可以把公式拆解成两个部分：

$$\tilde{Q}^{\text{ret}}(\boldsymbol{s}_t, \boldsymbol{a}_t) = \boldsymbol{r}_t + \gamma Q^{\text{ret}}(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) \times (1 - d_t) \quad （第一部分）$$

$$Q^{\text{ret}}(\boldsymbol{s}_t, \boldsymbol{a}_t) = E_{\boldsymbol{a}\sim\mu}[\rho_{t+1}(\tilde{Q}^{\text{ret}}(\boldsymbol{s}_t, \boldsymbol{a}_t) - Q_\mu(\boldsymbol{s}_t, \boldsymbol{a}_t))] + V_\pi(\boldsymbol{s}_t) \quad （第二部分）$$

实际上第二部分的计算是为第一部分服务的，最终要保留的是第一部分的结果。了解了这个原理，就可以把这两部分应用到下面的代码中。以下是对代码的详细介绍：

```
# 对rho进行限定，使其最小值为1
# 代码中的batch_to_seq可以理解为将一个原本为(nenv, nsteps)的矩阵变换为
一个长度为nsteps的序列，里面的每一个元素都是长度为nenv的数组
rho_bar = batch_to_seq(tf.minimum(1.0, rho_i), nenvs, nsteps, True)
rs = batch_to_seq(R, nenvs, nsteps, True)
ds = batch_to_seq(D, nenvs, nsteps, True)
q_is = batch_to_seq(q_i, nenvs, nsteps, True)
vs = batch_to_seq(v, nenvs, nsteps + 1, True)
# 我们将最后一个时刻的值函数直接设置为模型计算得到的v
v_final = vs[-1]
qret = v_final
qrets = []
# 开始迭代计算
for i in range(nsteps - 1, -1, -1):
    # 对应第一部分的公式
    qret = rs[i] + gamma * qret * (1.0 - ds[i])
    qrets.append(qret)
    # 对应第二部分的公式
    qret = (rho_bar[i] * (qret - q_is[i])) + vs[i]
# 将计算得到的值按顺序翻转
qrets = qrets[::-1]
# 重新变回维度为(nenvs, nsteps)的格式
qret = seq_to_batch(qrets, flat=True)
return qret
```

这样我们就完成了 Retrace 的计算过程。

11.2 ACER

本节我们介绍的方法叫作 Actor Critic with Experience Replay（简称 ACER），来自论文 *Sample Efficient Actor-Critic with Experience Replay*[3]，这是一种 Off-Policy 的 Actor-Critic 算法。由于 Off-Policy 在一些约定上与第 9 章及更前面的约定不同，同时也为了让读者对前面的内容做一个回顾，本节我们将重新对问题进行定义。

11.2.1 Off-Policy Actor-Critic

本节我们依然沿用第 6 章介绍的 MDP 框架。对于一个强化学习任务，我们用 $\boldsymbol{S}$ 表示状态集合，$\boldsymbol{A}$ 表示行动集合，那么状态转移概率分布可以表示为 P：$|\boldsymbol{S}|\times|\boldsymbol{S}|\times|\boldsymbol{A}|\to[0,1]$，在公式中我们用 $p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$ 表示由状态 $\boldsymbol{s}$、行动 $\boldsymbol{a}$ 转移到状态 $\boldsymbol{s}'$ 的概率，用 $\boldsymbol{r}$ 表示 Agent 与环境交互过程中获得的反馈与回报。γ 表示长期回报中未来回报对当前的打折率。状态价值可以用下面的公式表示：

$$V_\pi(\boldsymbol{s}_t)=E[\boldsymbol{r}_{t+1}+\cdots+\boldsymbol{r}_{t+T}|\boldsymbol{s}_t=\boldsymbol{s}]$$

这里我们假设任务从状态 $\boldsymbol{s}_t$ 出发，经过有限的时间长度 T，最终到达任务的终点。根据状态价值，我们还可以得出状态行动价值公式：

$$Q_\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)=\sum_{\boldsymbol{s}'\in\boldsymbol{S}}p(\boldsymbol{s}'|\boldsymbol{s}_t,\boldsymbol{a}_t)[\boldsymbol{r}_{t+1}+\gamma V_\pi(\boldsymbol{s}')]$$

Off-Policy 的策略梯度法涉及两个不同的策略。我们定义 μ 为交互的序列样本 $\{\boldsymbol{s}_0,\boldsymbol{a}_0,\boldsymbol{r}_0,\mu_0,\cdots\}$ 使用的策略，这些样本在交互后保存下来，并在未来的时刻应用于对某个策略 π 的训练中。

我们再定义状态出现的频率 $d^\mu(\boldsymbol{s})=\lim_{t\to\infty}p(\boldsymbol{s}_t=\boldsymbol{s}|\boldsymbol{s}_0,\mu)$，它表示了在遵循策略 μ 时，从任务的起始状态 $\boldsymbol{s}_0$ 出发，任意时刻到达状态 $\boldsymbol{s}$ 的概率之和。这样我们就可以定义目标：最大化从交互样本的每一状态出发的长期回报，令 θ 为策略 π 的参数，那么目标公式可以写作

$$J(\theta)=\sum_{\boldsymbol{s}\in\boldsymbol{S}}d^\mu(\boldsymbol{s})V_\pi(\boldsymbol{s})$$

如果我们的问题是一个 On-Policy 的问题，那么目标公式中的 $d^\mu(\boldsymbol{s})$ 将被换作 $d^\pi(\boldsymbol{s})$，如 9.1 节所示，我们已经了解过它的求解算法。这里使用与 9.1 节类似的求解方法，对

目标函数直接求导，可以得到

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta[\sum_{s\in S} d^\mu(s) \sum_{a\in A} \pi(a|s)Q_\pi(s,a)] \\ &= \sum_{s\in S} d^\mu(s) \sum_{a\in A} [\nabla_\theta \pi(a|s)Q_\pi(s,a) + \pi(a|s)\nabla_\theta Q_\pi(s,a)]\end{aligned}$$

对公式来说，$\nabla_\theta Q_\pi(s,a)$ 是一个比较难求解的项目，尤其是在它和 $d^\mu(s)$ 混合表示的情况下。因此，对 Off-Policy 的方法来说，我们可以忽略这一项，于是公式就变为

$$\nabla_\theta J(\theta) \simeq g(\theta) = \sum_{s\in S} d^\mu(s) \sum_{a\in A} [\nabla_\theta \pi(a|s)Q_\pi(s,a)]$$

实际上即使忽略掉那一部分的公式，我们依然可以在一定的条件下确保策略的提升。我们定义参数的更新公式为

$$\theta' \leftarrow \theta + \alpha g(\theta)$$

当 α 被限定为一个比较小的值时，例如存在一个 $\epsilon > 0$，使得对所有的 $\alpha < \epsilon$，依然可以得到

$$J(\theta') \geqslant J(\theta)$$

这个结论要如何证明呢？我们知道现在的近似梯度 $g(\theta)$ 实际上求解了策略 π 的梯度。也就是说，在 θ 附近的区域内，我们可以沿着 $g(\theta)$ 的方向使策略得到提高。于是我们可以通过设置 α，满足 $\pi(a|s;\theta') \geqslant \pi(a|s;\theta)$，于是就可以得到

$$\begin{aligned}J(\theta) &= \sum_{s\in S} d^\mu(s) \sum_{a\in A} \pi(a|s;\theta)Q_\pi(s,a) \\ &\leqslant \sum_{s\in S} d^\mu(s) \sum_{a\in A} \pi(a|s;\theta')Q_\pi(s,a)\end{aligned}$$

进一步替换，还可以得到

$$\begin{aligned}&= \sum_{s\in S} d^\mu(s) \sum_{a\in A} \pi(a|s;\theta') \sum_{s'\in S} p(s'|s_t,a_t)[r_{t+1} + \gamma V_\pi(s')] \\ &= \sum_{s\in S} d^\mu(s) \sum_{a\in A} \pi(a|s;\theta') \sum_{s'\in S} p(s'|s_t,a_t)[r_{t+1} + \gamma[\sum_{a'\in A} \pi(a'|s';\theta)Q_\pi(s',a')]] \\ &\leqslant \sum_{s\in S} d^\mu(s) \sum_{a\in A} \pi(a|s;\theta') \sum_{s'\in S} p(s'|s_t,a_t)[r_{t+1} + \gamma[\sum_{a'\in A} \pi(a'|s';\theta')Q_\pi(s',a')]]\end{aligned}$$

仅供非商业用途或交流学习使用

这样不断地展开，就可以得到

$$\begin{aligned}&\leqslant \sum_{s\in S} d^{\mu}(s) \sum_{a\in A} \pi(a|s;\theta')Q_{\pi_{\theta'}}(s,a)\\&= J(\theta')\end{aligned}$$

基于这样的证明，我们发现这个简化的梯度同样可以对目标函数进行优化，因此我们可以选择这个方法。将梯度公式进一步变换，可以得到更易于计算的形式，用 β 表示 d^{μ}，可以得到

$$\begin{aligned}g(\theta) &= \sum_{s\in S} \beta(s) \sum_{a\in A} [\nabla_{\theta}\pi(a|s)Q_{\pi}(s,a)]\\&= E_{s\sim\beta}[\sum_{a\in A} \nabla_{\theta}\pi(a|s)Q_{\pi}(s,a)]\\&= E_{s\sim\beta}[\sum_{a\in A} \mu(a|s)\frac{\pi(a|s)}{\mu(a|s)}\frac{\nabla_{\theta}\pi(a|s)}{\pi(a|s)}Q_{\pi}(s,a)]\\&= E_{s\sim\beta,a\sim\mu(\cdot|s)}[\rho(s,a)\nabla_{\theta}\log\pi(a|s)Q_{\pi}(s,a)]\end{aligned}$$

其中 $\rho(s,a)=\frac{\pi(a|s)}{\mu(a|s)}$，表示了两个策略的比值。现在公式中的状态和行动完全遵循策略 μ，我们就可以采用蒙特卡罗法进行采样，并使用上面的公式进行梯度计算。

看上去核心工作已经完成，而实际上同 11.1 节类似，两个策略的比值可能对训练造成不好的影响。如果两个策略的概率比值差异很大，那么梯度更新的步长也会很大，模型就会产生较大的波动，接下来介绍的 ACER 算法就是要解决这方面的问题。

11.2.2 ACER 算法

在 11.1 节，我们介绍了 Retrace 算法，它通过对新旧策略比值进行限定减少重要性采样带来的不稳定性。ACER 算法也要使用类似的方式，首先，我们将公式分解成 On-Policy 的部分和 Off-Policy 的部分。公式计算如下所示：

$$\begin{aligned}\boldsymbol{g}^{\text{marg}} &= E_{s_t\sim\beta,a_t\sim\mu}[\rho_t\nabla_{\theta}\log\pi_{\theta}(a_t|s_t)Q^{\pi}(s_t,a_t)]\\&= E_{s_t\sim\beta,a_t\sim\mu}[((1-\frac{c}{\rho_t}+\frac{c}{\rho_t})\rho_t\nabla_{\theta}\log\pi_{\theta}(a_t|s_t)Q^{\pi}(s_t,a_t)]\\&= E_{s_t\sim\beta}[E_{a_t\sim\mu}[((1-\frac{c}{\rho_t}+\frac{c}{\rho_t})\rho_t\nabla_{\theta}\log\pi_{\theta}(a_t|s_t)Q^{\pi}(s_t,a_t)]]\\&= E_{s_t\sim\beta}[E_{a_t\sim\mu}[\frac{c}{\rho_t}\rho_t\nabla_{\theta}\log\pi_{\theta}(a_t|s_t)Q^{\pi}(s_t,a_t)]\end{aligned}$$

仅供非商业用途或交流学习使用

$$
\begin{aligned}
&+ E_{\boldsymbol{a}_t\sim\mu}[(1-\frac{c}{\rho_t})\rho_t\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q^\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)]] \\
&= E_{\boldsymbol{s}_t\sim\beta}[E_{\boldsymbol{a}_t\sim\mu}[\frac{c}{\rho_t}\rho_t\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q^\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)] \\
&+ E_{\boldsymbol{a}_t\sim\pi}[(\frac{\rho_t-c}{\rho_t})\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q^\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)]]
\end{aligned}
$$

如果我们用一个经过限定的比率 $\tilde{\rho}_t$ 替换其中一部分的 ρ_t，同时限定 $\tilde{\rho}_t=\max(\rho_t,c)$，就有

$$
\begin{aligned}
&\frac{c}{\tilde{\rho}_t}=\min(\frac{c}{\rho_t},1),\quad \frac{c}{\tilde{\rho}_t}\rho_t=\min(c,\rho_t) \\
&\tilde{\rho}_t-c=\max(\rho_t-c,0),\quad \frac{\tilde{\rho}_t-c}{\rho_t}=\max(\frac{\rho_t-c}{\rho_t},0)
\end{aligned}
$$

我们就可以将上面的公式替换为 ACER 算法的公式：

$$
\begin{aligned}
&= E_{\boldsymbol{s}_t\sim\beta}[E_{\boldsymbol{a}_t\sim\mu}[\min(c,\rho_t)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q^\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)] \\
&+ E_{\boldsymbol{a}_t\sim\pi}[(\max(\frac{\rho_t-c}{\rho_t},0)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q^\pi(\boldsymbol{s}_t,\boldsymbol{a}_t)]]
\end{aligned}
$$

这个公式中的两个项目对算法起到不同的作用。第一项被称为截断的重要性采样（Truncated Importance Sampling），和 Retrace 算法的思想类似，它限制了重要性采样的比率上限，我们可以确保模型不会造成太大的波动。第二项被称为偏差纠正项（Bias Correction for the Truncation），作为前面一项的纠正项，它可以保证算法是无偏的，这样算法就在偏差和方差之间得到了一定的平衡。实际上，只有 $\rho_t>c$ 时，第二项才会发挥作用，而此时第一项的比率被限定为 c，这时第二项会对第一项的限制做一定的补充，从而保证算法没有较大的偏差。

完成了这一步的变换，下面就要对值函数做一定的替换。对上面公式的第一项，由于是 Off-Policy 的期望计算，可以用 11.1 节介绍的 Retrace(λ) 算法计算价值估计值 $Q^{\text{ret}}(\boldsymbol{s}_t,\boldsymbol{a}_t)$，而第二项由于是 On-Policy 的价值计算，因此直接使用价值模型的估计值即可，模型的参数为 θ_v。这样公式就变成了下面的形式：

$$
\begin{aligned}
&= E_{\boldsymbol{s}_t\sim\beta}[E_{\boldsymbol{a}_t\sim\mu}[\min(c,\rho_t)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q^{\text{ret}}(\boldsymbol{s}_t,\boldsymbol{a}_t)] \\
&+ E_{\boldsymbol{a}_t\sim\pi}[(\max(\frac{\rho_t-c}{\rho_t},0)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)Q_{\theta_v}(\boldsymbol{s}_t,\boldsymbol{a}_t)]]
\end{aligned}
$$

最后一步是 Actor-Critic 算法中常见的步骤，为算法添加一个 Baselines，以降低目标函数的方差，我们可以用价值模型估计当前的状态值函数 $V(\boldsymbol{s})$，最终的模型变为

仅供非商业用途或交流学习使用

$$= E_{s_t\sim\beta}[E_{a_t\sim\mu}[\min(c,\rho_t)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)(Q^{\text{ret}}(\boldsymbol{s}_t,\boldsymbol{a}_t)-V(\boldsymbol{s}_t))]$$
$$+ E_{a_t\sim\pi}[(\max(\frac{\rho_t-c}{\rho_t},0)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)(Q_{\theta_v}(\boldsymbol{s}_t,\boldsymbol{a}_t)-V_{\theta_v}(\boldsymbol{s}_t))]]$$

完成了上面对目标函数的定义，下面就来介绍求解的具体过程。10.1 节已经介绍了 TRPO 方法，它能够比较好地解决 Actor-Critic 算法中模型效果波动较大的问题，于是我们也采用同样思想的方法进行求解。但是由于两个算法要解决的问题不同，同时希望加快模型训练的速度，ACER 算法采取了两个改变：采用更保守的参数更新方式和相对简单的目标函数。

第一个改变是参数更新方式。ACER 维护了一个滑动平均的策略网络，每一次模型更新时，新的参数仅会以一个很小比例进行更新。这样我们就可以确保每一次更新后的参数和之前的参数保持较近的距离。这样算法可以以较小的代价实现类似 TRPO 约束的效果。我们令平均策略的参数为 θ_a，优化后的策略参数为 θ，那么平均策略的更新公式为

$$\theta_a \leftarrow \alpha\theta + (1-\alpha)\theta_a$$

第二个改变是目标函数。ACER 目标函数中计算一阶梯度的部分与 TRPO 比较相似，只要将前面得到的公式求导就可以得到

$$\hat{\boldsymbol{g}}_t^{\text{acer}} = \min(c,\rho_t)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)[Q^{\text{ret}}(\boldsymbol{s}_t,\boldsymbol{a}_t)-V_{\theta_v}(\boldsymbol{s}_t)]$$
$$+ E_{a_t\sim\pi}[(\max(\frac{\rho_t-c}{\rho_t},0)\nabla_\theta \log\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)(Q_{\theta_v}(\boldsymbol{s}_t,\boldsymbol{a}_t)-V_{\theta_v}(\boldsymbol{s}_t))]$$

ACER 的约束条件与 TRPO 不同，没有使用二阶导数做约束，而是使用了 KL 散度的一阶导数，这样目标函数可以变为下面的形式：

$$\text{minimize}_{\boldsymbol{z}} \frac{1}{2}\|\hat{\boldsymbol{g}}_t^{\text{acer}} - \boldsymbol{z}\|_2^2$$
$$\text{s.t.} \quad \nabla_{\pi_\theta(x_t)} D_{\text{KL}}[\pi_{\theta_a}\|\pi_\theta]^{\text{T}}\boldsymbol{z} < \boldsymbol{\delta}$$

可以看出，这个目标函数虽然也拥有基于 KL 散度的约束条件，但是它与 TRPO 的形式完全不同。经过这样的变换，我们不再需要计算复杂的 Fisher 信息矩阵。最终的目标是求解的参数更新量 $\boldsymbol{z}$，我们希望它在满足约束条件的同时尽可能地靠近 $\hat{\boldsymbol{g}}_t^{\text{acer}}$ 值。

上面提到的对目标函数的两个改变实际上为更新量的求解提供了很大的便利。为了后面求解公式时的便利，我们做如下的变量替换：

$$\hat{\boldsymbol{g}}_t^{\text{acer}} = \boldsymbol{g}$$

仅供非商业用途或交流学习使用

$$\nabla_{\phi_\theta(x_t)} D_{\mathrm{KL}}[\pi_{\theta_a} \| \pi_\theta] = \boldsymbol{k}$$

这个目标函数实际上是一个二次规划的问题。由于问题中目标函数的二阶导为半正定矩阵（在这个问题中，二阶导实际上是一个对角阵），我们可以证明这个问题满足 KKT 条件，它的对偶问题具有强对偶的特性，原问题和它的对偶问题的解相同。于是我们就可以构造这个问题的拉格朗日乘子法形式：

$$\mathrm{minimize}_{\boldsymbol{z}} \mathrm{H}(\boldsymbol{z}, \lambda) = \frac{1}{2}\|\boldsymbol{g} - \boldsymbol{z}\|_2^2 + \lambda(\boldsymbol{k}^{\mathrm{T}}\boldsymbol{z} - \boldsymbol{\delta})$$

首先对 $\boldsymbol{z}$ 进行求导，可以得到

$$\frac{\partial \mathrm{H}(\boldsymbol{z}, \lambda)}{\partial \boldsymbol{z}} = \boldsymbol{z} - \boldsymbol{g} + \lambda \boldsymbol{k}$$

令导数为 0，可以得到 $\boldsymbol{z}$ 的最优值

$$\boldsymbol{z}^* = \boldsymbol{g} - \lambda \boldsymbol{k}$$

将其带回上面的拉格朗日公式中，可以得到

$$\begin{aligned}
\mathrm{H}(\lambda) &= \frac{1}{2}\lambda^2\|\boldsymbol{k}\|_2^2 + \lambda(\boldsymbol{k}^{\mathrm{T}}(\boldsymbol{g} - \lambda\boldsymbol{k}) - \boldsymbol{\delta}) \\
&= \frac{1}{2}\|\boldsymbol{k}\|_2^2\lambda^2 + \lambda\boldsymbol{k}^{\mathrm{T}}\boldsymbol{g} - \lambda^2\|\boldsymbol{k}\|_2^2 - \lambda\boldsymbol{\delta} \\
&= -\frac{1}{2}\|\boldsymbol{k}\|_2^2\lambda^2 + (\boldsymbol{k}^{\mathrm{T}}\boldsymbol{g} - \boldsymbol{\delta})\lambda
\end{aligned}$$

对其进行求导，可以得到

$$\frac{\partial \mathrm{H}(\lambda)}{\partial \lambda} = -\|\boldsymbol{k}\|_2^2\lambda + (\boldsymbol{k}^{\mathrm{T}}\boldsymbol{g} - \boldsymbol{\delta})$$

令导数为 0，可以得到

$$\lambda^* = \frac{\boldsymbol{k}^{\mathrm{T}}\boldsymbol{g} - \boldsymbol{\delta}}{\|\boldsymbol{k}\|_2^2}$$

因为朗格朗日的乘子被限定为不小于 0，所以 λ^* 不小于 0，将其带回前面 $\boldsymbol{z}^*$ 表示的公式，可以得到

$$\boldsymbol{z}^* = \boldsymbol{g} - \max(0, \frac{\boldsymbol{k}^{\mathrm{T}}\boldsymbol{g} - \boldsymbol{\delta}}{\|\boldsymbol{k}\|_2^2})\boldsymbol{k}$$

仅供非商业用途或交流学习使用

从结果可以看出，当约束条件可以得到满足时，最终的更新值就是梯度 $\boldsymbol{g}$；如果约束无法满足，那么 $\boldsymbol{z}$ 就会沿着 $\boldsymbol{k}$ 的方向进行一定的缩减，使更新量满足约束，这样就得到了我们想要的更新量。

11.2.3 ACER 的实现

ACER 的完整算法流程如下所示，流程主要包含两个子过程，一个是采用 On-Policy 的优化，也就是基于当前的策略与环境交互采集样本；另一个是采用 Off-Policy 的优化，直接使用保存起来的样本进行训练。这个训练流程和 DQN 的训练方法十分类似，ACER 和 DQN 都需要定期使用新的策略采集样本，以保证 Replay Buffer 中采样样本的策略和当前的策略差距不大。

由于在 ACER 中需要计算两个策略的概率比率，因此在 Replay Buffer 中需要将采样策略的行动概率值保存，因此 Replay Buffer 中就需要额外保存这个信息。

算法 ACER 算法的主体

Repeat

Call ACER (on-policy = True)

for $i \in \{1, \cdots, n\}$ **do**

Call ACER (on-policy = False)

end for

until 最大的迭代轮数满足

子算法 离散行动的 ACER (on-policy)

重置参数梯度：$\mathrm{d}\theta \leftarrow 0$，$\mathrm{d}\theta_v \leftarrow 0$

初始化参数：$\theta' \leftarrow \theta$，$\theta'_v \leftarrow \theta_v$

If on-policy==True:

利用 policy 和环境交互得到 $\{\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{r}_t, \mu_t, d_t\}_{t=1}^{k}$，并将其放入 Replay Buffer，其中：$\mu(\cdot|\boldsymbol{s}_i) \leftarrow \pi(\boldsymbol{a}|\boldsymbol{s}_i; \theta')$

else:

从 Replay Buffer 中采样得到 $\{\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{r}_t, \mu_t, d_t\}_{t=1}^{k}$

end if

for $i \in \{0, \cdots, k\}$ **do**

计算：$\pi(\boldsymbol{a}|\boldsymbol{s}_t; \theta')$、$Q(\boldsymbol{s}_t, \cdot; \theta'_v)$ 和 $\pi(\boldsymbol{a}|\boldsymbol{s}_t; \theta_a)$

$\bar{\rho}_t \leftarrow \min\{1, \frac{\pi(\boldsymbol{a}_t|\boldsymbol{s}_t;\theta')}{\mu(\boldsymbol{a}_t|\boldsymbol{s}_t)}\}$

end for

仅供非商业用途或交流学习使用

$$Q^{\text{ret}} \leftarrow \begin{cases} 0 & \text{终止状态} s_t \\ \sum_a Q(s_t, a|\theta'_v)\pi(a|s_t;\theta') & \text{其他情况} \end{cases}$$

for $t \in \{k-1, \cdots, 0\}$ **do**

$Q^{\text{ret}} \leftarrow r_i + \gamma Q^{\text{ret}}$

$V_t \leftarrow \sum_a Q(s_t, a; \theta'_v)\pi(a|s_t;\theta')$

计算 Trust Region 目标需要的部分公式：

$g \leftarrow \min(c, \rho_t)\nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t)[Q^{\text{ret}}(s_t, a_t) - V_{\theta'_v}(s_t)] +$

$\sum_a [(\max(\frac{\rho_t - c}{\rho_t}, 0)\nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t)(Q_{\theta'_v}(s_t, a_t) - V_{\theta'_v}(s_t))]$

$k \leftarrow \nabla_{\pi_{\theta'}} D_{\text{KL}}[\pi_{\theta_a} \| \pi_{\theta'}]$

计算策略参数 θ' 的更新量：$\mathrm{d}\theta' \leftarrow \mathrm{d}\theta + g - \max(0, \frac{k^{\mathrm{T}} g - \delta}{\|k\|_2^2})k$

计算价值参数 θ'_v 的更新量：$\mathrm{d}\theta'_v \leftarrow \mathrm{d}\theta'_v + \nabla_{\theta'_v}(Q^{\text{ret}} - Q_{\theta'_v}(s_t, a)^2$

更新 Retrace 目标：$Q^{\text{ret}} \leftarrow \bar{\rho}_t(Q^{\text{ret}} - Q(s_t, a_t; \theta'_v)) + V_t$

end for

将 $\mathrm{d}\theta'$ 和 $\mathrm{d}\theta'_v$ 更新到原始的参数 θ 和 θ_v 上

更新平均策略参数：$\theta_a \leftarrow \alpha\theta_a + (1-\alpha)\theta$

介绍完 ACER 的算法流程，再介绍它在 Baselines 中的实现。这部分代码在 baselines/acer 项目中，核心代码在 acer_simple.py 中。由于涉及的内容比较多，模型架构相对复杂，如图 11-1 和图 11-2 所示。

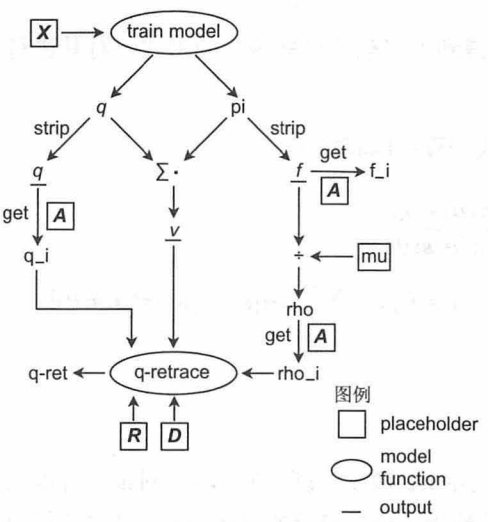


图 11-1 ACER 的流程图 1

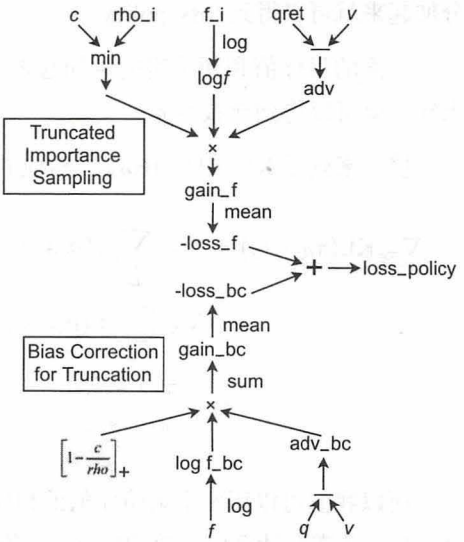


图 11-2 ACER 的流程图 2

仅供非商业用途或交流学习使用

图 11-1 所示为 Q^{ret} 的计算过程。算法的模型部分和 A2C 的模型有些不同，它要输出 $\pi(\boldsymbol{a}|\boldsymbol{s})$ 和 $q(\boldsymbol{s},\boldsymbol{a})$ 两个值，在图中分别用 pi 和 q 表示。这两个值进行乘加操作，可以得到价值 $\boldsymbol{v}$。代码中有一个 strip 操作，用于将数据转换成算法中易于处理的形式，pi 变形成 f，并与 Replay Buffer 中存储的代表 $\mu(\boldsymbol{a}|\boldsymbol{s})$ 的 mu 相除，得到策略的比值 rho。

当我们知道了回报 $\boldsymbol{R}$，终止状态 $\boldsymbol{D}$，价值 $\boldsymbol{q}$、$\boldsymbol{v}$，策略比值 rho 后，就可以利用 Retrace 方法计算出 Q^{ret} 值 qret。

图 11-2 所示为 ACER 计算公式中的两部分。首先是截断的重要性采样部分，它由 Q^{ret} 和价值的估计值的差、策略的对数似然和比率阶段值三部分组成，公式为

$$\min(c,\rho_t)\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{a}_t|\boldsymbol{s}_t)[Q^{\text{ret}}(\boldsymbol{s}_t,\boldsymbol{a}_t)-V_{\theta'_v}(\boldsymbol{s}_t)]$$

其中后两项的计算结果分别为 logf 和 adv，由于这三项中只有第二项参与梯度计算，因此第一、第三项需要被 tf.stop_gradient 包含，以关闭它们的反向计算。

ACER 公式的第二部分，也就是偏差纠正部分，同样是由比率值、对数似然和公式及价值差组成，对应的公式为

$$\sum_{\boldsymbol{a}}(\max(\frac{\rho_t-c}{\rho_t},0)\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{a}_t|\boldsymbol{s}_t)(Q_{\theta'_v}(\boldsymbol{s}_t,\boldsymbol{a}_t)-V_{\theta'_v}(\boldsymbol{s}_t))$$

其中后两项的计算结果分别为 logf_bc 和 adv_bc，这个公式同样只有第二项参与梯度计算，因此处理方式与前面一致。这样得到了两部分的 loss 值：loss_f 与 loss_bc，将两部分加起来就可以得到 loss_policy。

上面的目标值和新策略的熵加起来，就得到了完整的目标函数的形式，对其进行求导，就可以得到导数值 g。

接下来则是 KL 散度的梯度，对应的公式计算可以转换为

$$\begin{aligned}\nabla_{\pi_\theta}\text{KL}(\pi_{\theta_a}\|\pi_\theta)&=\nabla_{\pi_\theta}\sum_{\boldsymbol{a}}\pi(\boldsymbol{a}|\boldsymbol{s};\theta_a)\log\frac{\pi(\boldsymbol{a}|\boldsymbol{s};\theta_a)}{\pi(\boldsymbol{a}|\boldsymbol{s};\theta)}\\&=\nabla_{\pi_\theta}[\sum_{\boldsymbol{a}}\pi(\boldsymbol{a}|\boldsymbol{s};\theta_a)\log\pi(\boldsymbol{a}|\boldsymbol{s};\theta_a)-\sum_{\boldsymbol{a}}\pi(\boldsymbol{a}|\boldsymbol{s};\theta_a)\log\pi(\boldsymbol{a}|\boldsymbol{s};\theta)]\\&=-\frac{\pi(\boldsymbol{a}|\boldsymbol{s};\theta_a)}{\pi(\boldsymbol{a}|\boldsymbol{s};\theta)}\end{aligned}$$

所以我们可以用两个策略比值的相反数表示 KL 散度梯度。在完成了对策略梯度的计算后，还需要计算值函数的梯度，而值函数使用了比较简单的目标函数，计算比较简单，具体过程请读者自行阅读。

可以看出，只要能够理解 Retrace 算法和 ACER 中形似 TRPO 目标的目标函数，整体的算法流程是比较容易理解的。前面我们介绍的只是离散行动的解法，关于连续行动空间的解法，欢迎读者阅读论文进行了解，其主体思想与离散方法基本一致，只是在一些细节上有所不同。

11.3 DPG

11.2 节从 Experience Replay 的角度实现了 Actor-Critic 方法，本节要介绍另外一个 Off-Policy 的算法，这个方法被称为 Deterministic Policy Gradient（简称 DPG），这个算法经过几篇论文的不断打磨改进，效果变得越来越好。这个算法的名称看上去和 Policy Gradient 相似，但其实际计算过程并不相同，从名字中可以看出，这个方法使用了确定的策略，这不同于之前提到的输出分布的策略形式。下面我们就来看看算法的计算过程。

11.3.1 连续空间的策略优化

在前面的章节中，我们主要介绍了离散行动空间的任务，例如 Atari 游戏，由于游戏手柄的操作形式有限，我们可以使用探索算法尽可能地将状态行动枚举出来。对于行动连续的任务，想要枚举所有的行动变得更困难，而如何将所有可行的行动逐一尝试出来也变得不那么可能。我们先回顾前面介绍过的两种强化学习算法的计算公式。首先是 DQN 算法的更新公式：

$$\Delta q(\boldsymbol{s},\boldsymbol{a}) = r(\boldsymbol{s}') + \max_{\boldsymbol{a}'} q^{T-1}(\boldsymbol{s}',\boldsymbol{a}')$$

DQN 算法遵循了泛化策略梯度的思想，先完成策略评估的工作，再进行策略改进。这个过程需要计算出下一时刻状态下所有行动的价值，并从中选出最优的行动价值。如果行动数量是有限的，那么这个公式比较好计算；如果行动空间是连续的，我们该如何找出价值最优的行动呢？这就成为了一个问题。

接下来是策略梯度法，它的核心计算公式为

$$\nabla_\theta J(\theta) = E_{\boldsymbol{\tau}\sim\pi_\theta(\boldsymbol{\tau})}[(\sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})(\sum_{t=0}^{T} r(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})]$$

策略梯度法由于直接对轨迹的价值期望求导，因此它不需要完成最优行动选取的

操作。因此连续型行动空间的问题可以使用策略梯度算法求解，但是策略梯度法是一个 On-Policy 的方法，它非常依赖与环境交互的过程，而 DQN 方法直接对值函数进行优化，可以使用 Off-Policy 的方法进行训练。所以，如果我们想在连续行动空间使用 Off-Policy 算法进行优化，可以考虑结合两种算法的特点，创造出一种全新的算法。不同于 11.2 节使用重要性采样进行求解，本节介绍的 DPG 算法将采用另外一种方式。

在与环境交互时，DQN 算法一般使用 $\epsilon-$greedy 的策略，策略梯度是从一个概率分布中采样得到的，而 DPG 的交互方式结合了前面两种算法。从形式上看，DPG 使用了 $\epsilon-$greedy 的策略，以一定的概率使用随机策略，而在剩下的情况下使用最优行动；从策略产生的行动上看，DPG 将先得到一个确定的行动，这个行动由确定的策略得到，不需要从概率分布中采样，相当于当前状态下的最优行动。如果决定使用随机策略，那么就在求出的确定行动基础上加上一定的噪声，反之则没有噪声。

虽然确定策略的思想和 DQN 相近，但实际上，DPG 也可以看作是策略梯度法的一种特殊情况。我们知道随机策略梯度的输出是行动分布形式，对于离散行动空间，模型输出的是一个 Category 的分布，也就是每一个取值的概率；而对于连续行动空间，我们一般会输出一个高斯分布，其中一部分值表示分布的均值，另一部分值表示分布的方差，然后我们可以使用这些分布的参数采样出行动值。DPG 的输出也可以想象成一个连续的分布，只不过这个分布的方差为 0。这样我们就把 DPG 和策略梯度法统一起来了，前面介绍的策略梯度法也可以称为随机策略梯度法。

下面我们就利用这个思路建立目标。DPG 的算法本身还是采用策略梯度法的方法，直接对轨迹价值求导。但是由于策略产生的行动是确定的，于是行动就可以直接被替换为策略函数，公式将变为下面的形式：

$$\nabla_\theta v_\mu(\boldsymbol{s}) = \nabla_\theta[q_\mu(\boldsymbol{s}, \mu(\boldsymbol{s}))], \forall \boldsymbol{s} \in \boldsymbol{S}$$

其中 μ 表示生成确定行动的策略函数。将上面的公式进一步推导，利用链式法则可以得到

$$= \nabla_{\mu(\boldsymbol{s})} q_\mu(\boldsymbol{s}, \mu(\boldsymbol{s})) \nabla_\theta \mu(\boldsymbol{s})$$

同随机策略梯度法对比，这个梯度公式中没有了与行动相关的期望求解项。下面我们就围绕这个公式进行计算。

11.3.2 策略模型参数的一致性

在进一步介绍确定策略的目标推导前，我们需要简单介绍一个概念：策略模型参

数的一致性。在随机策略梯度方法的介绍过程中，我们曾提到直接使用一个函数来近似状态行动价值，即定义一个函数 $f_w : \boldsymbol{S} \times \boldsymbol{A} \to \boldsymbol{R}$，用来近似真实的值函数 Q^π，函数的参数为 w。如果我们用 L2 损失作为近似函数 f_w 和真实价值 Q^π 的损失函数，那么近似函数的梯度为

$$\nabla_w \text{Loss} \propto \nabla_w E_{\boldsymbol{s},\boldsymbol{a}}[\hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a}) - f_w(\boldsymbol{s},\boldsymbol{a})]^2 = E_{\boldsymbol{s},\boldsymbol{a}\sim\boldsymbol{\tau}}[(\hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a}) - f_w(\boldsymbol{s},\boldsymbol{a}))\nabla_w f_w(\boldsymbol{s},\boldsymbol{a})]$$

其中 $\hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a})$ 是状态行动的目标价值。如果函数 f_w 最终收敛，即下面的公式成立：

$$\sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \pi(\boldsymbol{s},\boldsymbol{a})[\hat{Q}^\pi(\boldsymbol{s},\boldsymbol{a}) - f_w(\boldsymbol{s},\boldsymbol{a})]\nabla_w f_w(\boldsymbol{s},\boldsymbol{a}) = 0$$

那么，在论文 *Policy Gradient Methods for Reinforcement Learning with Function Approximation*[6] 中，作者证明了，当 f_w 满足上面的公式，同时满足和策略函数参数的某种一致性，即 f_w 关于参数的梯度等于策略的对数似然关于参数的梯度：

$$\nabla_w f_w(\boldsymbol{s},\boldsymbol{a}) = \nabla_\theta \log\pi(\boldsymbol{s},\boldsymbol{a}) = \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) \frac{1}{\pi(\boldsymbol{s},\boldsymbol{a})}$$

那么，在计算梯度时我们就可以用 f_w 代替 Q^π，并得到相同的结果。具体的计算过程为

$$\begin{aligned}
\nabla_\theta v^\pi(\boldsymbol{s}) &= \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) Q^\pi(\boldsymbol{s},\boldsymbol{a}) - 0 \\
&= \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) Q^\pi(\boldsymbol{s},\boldsymbol{a}) \\
&\quad - \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \pi(\boldsymbol{s},\boldsymbol{a})[Q^\pi(\boldsymbol{s},\boldsymbol{a}) - f_w(\boldsymbol{s},\boldsymbol{a})]\nabla_w f_w(\boldsymbol{s},\boldsymbol{a}) \\
&= \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) Q^\pi(\boldsymbol{s},\boldsymbol{a}) \\
&\quad - \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \pi(\boldsymbol{s},\boldsymbol{a})[Q^\pi(\boldsymbol{s},\boldsymbol{a}) - f_w(\boldsymbol{s},\boldsymbol{a})]\nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) \frac{1}{\pi(\boldsymbol{s},\boldsymbol{a})} \\
&= \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) Q^\pi(\boldsymbol{s},\boldsymbol{a}) \\
&\quad - \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a})[Q^\pi(\boldsymbol{s},\boldsymbol{a}) - f_w(\boldsymbol{s},\boldsymbol{a})] \\
&= \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a})[Q^\pi(\boldsymbol{s},\boldsymbol{a}) - Q^\pi(\boldsymbol{s},\boldsymbol{a}) + f_w(\boldsymbol{s},\boldsymbol{a})] \\
&= \sum_{\boldsymbol{s}} d^\pi(\boldsymbol{s}) \sum_{\boldsymbol{a}} \nabla_\theta \pi(\boldsymbol{s},\boldsymbol{a}) f_w(\boldsymbol{s},\boldsymbol{a})
\end{aligned}$$

上面定理实现的关键是参数的一致性，这一点容易做到吗？由于策略模型的计算结果是对数似然概率，所以这样的形式很容易被满足。我们令策略模型是一个最简单的线性模型，当模型经过 Softmax 层计算后，可以得到

$$\pi(\boldsymbol{s},\boldsymbol{a})=\frac{\mathrm{e}^{\theta^{\mathrm{T}}\phi_{\boldsymbol{s},\boldsymbol{a}}}}{\sum_{\boldsymbol{b}}\mathrm{e}^{\theta^{\mathrm{T}}\phi_{\boldsymbol{s},\boldsymbol{b}}}}$$

对其取对数求导，可以得到

$$\begin{aligned}
\nabla_w f_w(\boldsymbol{s},\boldsymbol{a}) &= \nabla_\theta \log\pi(\boldsymbol{s},\boldsymbol{a}) \\
&= \nabla_\theta \log[\exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},\boldsymbol{a}}]-\sum_b \exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]] \\
&= \nabla_\theta[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},\boldsymbol{a}}]-\nabla_\theta \log[\sum_b \exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]] \\
&= \phi_{\boldsymbol{s},\boldsymbol{a}}-\frac{1}{\sum_b \exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]}\nabla_\theta[\sum_b \exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]] \\
&= \phi_{\boldsymbol{s},\boldsymbol{a}}-\frac{1}{\sum_b \exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]}\sum_b \exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]\phi_{\boldsymbol{s},b} \\
&= \phi_{\boldsymbol{s},\boldsymbol{a}}-\sum_b \frac{\exp[\theta^{\mathrm{T}}\phi_{\boldsymbol{s},b}]}{\sum_b \exp[\theta^{T}\phi_{\boldsymbol{s},b}]}\phi_{\boldsymbol{s},b} \\
&= \phi_{\boldsymbol{s},\boldsymbol{a}}-\sum_b \pi(\boldsymbol{s},b)\phi_{\boldsymbol{s},b} \\
&= \phi_{\boldsymbol{s},\boldsymbol{a}}-\boldsymbol{E}_{\boldsymbol{a}\sim\pi(\boldsymbol{s})}[\phi_{\boldsymbol{s},\boldsymbol{a}}]
\end{aligned}$$

上式的第二项是一个期望项，在实际中它是一个常量，我们可以将其忽略，只要将输入数据进行归一化即可抵消掉这一项。因此为了满足参数的一致性，我们可以将价值估计函数的形式设定为

$$f_w(\boldsymbol{s},\boldsymbol{a})=w^{\mathrm{T}}\hat{\phi}_{\boldsymbol{s},\boldsymbol{a}}$$

实际上，神经网络同样可以以这样的形式表示出来。因此，采用神经网络同样可以达到这样的一致性效果。读者看到这里一定充满疑惑，这个证明看上去有些无聊，那么它的意义何在呢？对于 DPG，作者在论文中给出了类似的证明与结论。

给定一个值函数 $Q^w(\boldsymbol{s},\boldsymbol{a})$ 与确定的策略函数 $\mu_\theta(\boldsymbol{s})$，当下面两个条件满足时：

（1）$\nabla_{\boldsymbol{a}}Q^w(\boldsymbol{s},\boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}=\nabla_\theta\mu_\theta(\boldsymbol{s})^{\mathrm{T}}w$

（2）参数 w 是近似值函数的局部最优值，它使得下面的目标函数取得最小值：

$$\text{MSE}(\theta, w) = E[\epsilon(\boldsymbol{s}; \theta, w)^{\mathrm{T}} \epsilon(\boldsymbol{s}; \theta, w)]$$

其中

$$\epsilon(\boldsymbol{s}; \theta, w) = \nabla_{\boldsymbol{a}} Q^{w}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})} - \nabla_{\boldsymbol{a}} Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}$$

我们可以得到下面的定理，也就是说可以用值函数近似真实的价值并得到正确的策略梯度：

$$\nabla_\theta J_\beta(\theta) = E[\nabla_\theta \mu_\theta(\boldsymbol{s}) \nabla_{\boldsymbol{a}} Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}] = E[\nabla_\theta \mu_\theta(\boldsymbol{s}) \nabla_{\boldsymbol{a}} Q^{w}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}]$$

我们可以简单证明一下，根据前面的一致性定义，可以得到

$$\nabla_w \epsilon(\boldsymbol{s}; \theta, w) = \nabla_\theta \mu_\theta(\boldsymbol{s})$$

继续展开前面的公式可以得到当 w 等于局部最优时，目标函数的导数为 0：

$$\nabla_w \text{MSE}(\theta, w) = E[\nabla_w [\epsilon(\boldsymbol{s}; \theta, w)^{\mathrm{T}} \epsilon(\boldsymbol{s}; \theta, w)]] = 0$$

接着展开可以得到

$$E[2\epsilon(\boldsymbol{s}; \theta, w) \nabla_w \epsilon(\boldsymbol{s}; \theta, w)] = 0$$
$$E[\epsilon(\boldsymbol{s}; \theta, w) \nabla_\theta \mu_\theta(\boldsymbol{s})] = 0$$
$$E[(\nabla_{\boldsymbol{a}} Q^{w}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})} - \nabla_{\boldsymbol{a}} Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}) \nabla_\theta \mu_\theta(\boldsymbol{s})] = 0$$
$$E[\nabla_\theta \mu_\theta(\boldsymbol{s}) \nabla_{\boldsymbol{a}} Q^{w}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}] = E[\nabla_\theta \mu_\theta(\boldsymbol{s}) \nabla_{\boldsymbol{a}} Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}] = \nabla_\theta J_\beta(\theta)$$

于是得证。实际上通过前面大段的证明，我们可以得出一个结论：对于目标函数中的价值估计部分，我们可以使用一个值函数模型进行拟合，这样价值模型不需要遵从某个策略。这样我们就可以使用 Off-Policy 的方法进行计算了。

11.3.3 DDPG 算法

接下来介绍 DPG 的实现。实际上，DPG 的思想仅仅是使用确定的策略输出，算法并不包含其他的限定，例如是否为 On-Policy 或者 Off-Policy。对于 On-Policy 的 Deterministic Actor-Critc 算法，值函数为 $Q^{w}(\boldsymbol{s}, \boldsymbol{a})$，确定策略为 $\mu_\theta(\boldsymbol{s})$，我们可以建立如下目

标函数：

$$J(w) = \text{minimize}_w E_\pi[\frac{1}{2}(\boldsymbol{r}_t + \gamma Q^w(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) - Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t))^2]$$
$$J(\theta) = \text{maximize}_\theta E_\pi[Q^w(\boldsymbol{s}_t, \mu(\boldsymbol{s}_t))]$$

对其进行求解，可以得到

$$\Delta w = \alpha E_\pi[(\boldsymbol{r}_t + \gamma Q^w(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) - Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t))\nabla_w Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t)]$$
$$\Delta\theta = \alpha E_\pi[\nabla_w Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t)|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}\nabla_\theta \mu_\theta(\boldsymbol{s}_t)]$$

对于 Off-Policy 的算法，我们同样可以建立目标函数。由于我们使用了确定的策略，同时值函数不依赖任何策略，那么在计算时我们就不需要向随机策略那样进行重要性采样计算。假设样本来自策略 β，我们的目标函数为

$$J(w) = \text{minimize}_w E_\beta[\frac{1}{2}(\boldsymbol{r}_t + \gamma Q^w(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) - Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t))^2]$$
$$J(\theta) = \text{maximize}_\theta E_\beta[Q^w(\boldsymbol{s}_t, \mu(\boldsymbol{s}_t))]$$

对其进行求解，可以得到类似的结果：

$$\Delta w = \alpha E_\beta[(\boldsymbol{r}_t + \gamma Q^w(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) - Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t))\nabla_w Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t)]$$
$$\Delta\theta = \alpha E_\beta[\nabla_w Q^w(\boldsymbol{s}_t, \boldsymbol{a}_t)|_{\boldsymbol{a}=\mu_\theta(\boldsymbol{s})}\nabla_\theta \mu_\theta(\boldsymbol{s}_t)]$$

由于本章的主题是介绍 Off-Policy 的算法，因此接下来我们只关心这部分的算法。虽然 DPG 从名字上看属于策略梯度法的一种，但是它的求解过程有很多与 DQN 相近的地方，正如论文 *Continuous control with deep reinforcement learning*[6] 中介绍的，DQN 的优化过程中的两个常见的优化方法也被应用到了 Deep DPG 上。

1. Replay-Buffer

由于连续的序列中存在的相关性会使 DQN 的优化充满不稳定性，因此 DQN 采用了 Replay-Buffer，将一些采样样本收集起来，每次优化时从中随机取出一部分进行优化，就可以减少一些不稳定性。这个方法同样适用于 DPG。

2. Target Network

这也是 DQN 中常见的方法，通过设置一个不会频繁或大幅更新的模型，使模型计算的值函数在一定程度上减少波动，令计算更稳定。在论文中，作者介绍采用滑动平均的方法更新 Target Network：$\theta_{t+1} \leftarrow \tau\theta_t + (1-\tau)\theta'_t$，$\tau$ 一般设置为非常接近 1 的数，这样 Target 网络的参数 θ 不会发生太大的变化，每次只会受一点训练模型 θ' 的影响。这个更新方法和 11.2 节的 ACER 算法十分相似。

除了这两个方法外，作者还对模型的探索做了一些改变。由于 DPG 采用确定策略，如果它在与环境进行交互时只采用确定的策略，那么必然会导致对环境的探索不够充分，因此需要为策略增加一定的探索性。前面已经提到 DPG 可以使用基于 $\epsilon-$greedy 的探索方法，通过为确定的行动增加噪声提高探索性。在 DDPG 模型中，作者采用 Ornstein-Uhlenbeck 噪声增加模型的探索能力。Ornstein-Uhlenbeck 噪声是一种基于 Ornstein-Uhlenbeck 过程的随机变量，它可以被用于模拟与时间有关联的噪声数据。它的生成公式为

$$\mathrm{d}x_t = \theta(\mu - x_t) + \sigma W_t$$

其中 x 是要生成的数据，μ 是设定的随机变量的期望值，W 是一个由 Wiener 过程生成的随机变量，一般我们用一个简单的随机函数代替就可以，θ 和 σ 是随机过程中的参数，$\mathrm{d}x$ 是每一时刻数据的变化量，真正的采样值等于上一时刻的采样值加上求出的变化量。从公式中可以看出，每一时刻数据的变化量和当前时刻存在关联，公式右边的第一项将为随机数据提供朝向均值 μ 的变化，第二项才是常见的随机变化。

为了更直观地理解这个噪声生成器，我们举一个例子。设定 $\theta = 0.01$，$\mu = 0$，$\sigma = 0.02$，W 由均值为 0，方差为 1 的高斯随机函数生成。我们设定三个随机变量，它们的初始值为 2、0 和 -2，对其进行一定时间的采样，得到图 11-3 所示的结果。

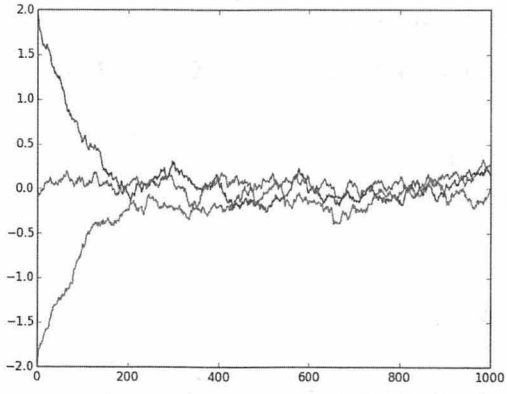


图 11-3 Ornstein-Uhlenbeck 的采样结果图

从结果可以看出，三个随机变量在波动中靠近了均值 0。这就可以确保噪声的增加不会对策略产生太大的影响，一旦噪声在某个维度造成了影响，一定会尽快给予补偿。

除了上面提到的一些改进方法，我们还可以从前面章节的内容中找到更多改进模型的灵感。第 8 章介绍了许多改进 DQN 的方法，这些方法都可以以某种形式应用到 Deep DPG 中。例如，Priority Replay Buffer、Pre-train from Demonstration 等。读者可以阅读论文 *Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards*，进一步了解关于这部分的改进过程。

完整的 DDPG 算法如下所示。

算法 DDPG 算法

初始化 critic 网络 $Q(\boldsymbol{s},\boldsymbol{a}|\theta^Q)$，actor 网络 $\mu(\boldsymbol{s}|\theta^\mu)$

用上面的两个网络初始化对应的目标网络 $Q'\leftarrow Q$，$\mu'\leftarrow\mu$

初始化 Replay Buffer：$\boldsymbol{R}$

for episode $=1,M$ **do**

初始化噪声分布 N

$\boldsymbol{s}_1$=env.reset()

for $t=1,T$ **do**

$\boldsymbol{a}_t=\mu(\boldsymbol{s}_t|\theta^\mu)+N_t$

$\boldsymbol{s}_{t+1}$, reward_t, terminate, _ = env.step($\boldsymbol{a}_t$)

$R.save((\boldsymbol{s}_t,\boldsymbol{a}_t,\boldsymbol{r}_t,\boldsymbol{s}_{t+1}))$

// 训练

$(\boldsymbol{s}_i,\boldsymbol{a}_i,\boldsymbol{r}_i,\boldsymbol{s}_{i+1})$=$R$.sample($N$)

$y_i=\boldsymbol{r}_i+\gamma Q'(\boldsymbol{s}_{i+1},\ \mu'(\boldsymbol{s}_{i+1}|\theta^{\mu'})|\theta^{Q'})$

根据 critic loss 更新 critic 网络：$L=\frac{1}{N}\sum_i(y_i-Q(\boldsymbol{s}_i,\boldsymbol{a}_i|\theta^Q))^2$

根据 actor 的梯度更新 actor 网络：

$\nabla_{\theta^\mu}J\simeq\frac{1}{N}\sum_i\nabla_{\boldsymbol{a}}Q(\boldsymbol{s},\boldsymbol{a}|\theta^Q)|_{\boldsymbol{s}=\boldsymbol{s}_i,\boldsymbol{a}=\mu(\boldsymbol{s}_i)}\nabla_{\theta^\mu}\mu(\boldsymbol{s}|\theta^\mu)|_{\boldsymbol{s}_i}$

更新目标网络：

$\theta^{Q'}\leftarrow\boldsymbol{\tau}\theta^Q+(1-\boldsymbol{\tau})\theta^{Q'}$

$\theta^{\mu'}\leftarrow\boldsymbol{\tau}\theta^\mu+(1-\boldsymbol{\tau})\theta^{\mu'}$

11.3.4 DDPG 的实现

在 Baselines 项目中的 baselines/ddpg 文件夹中，可以看到 DDPG 的实现。文件夹中各文件的含义如下所示。

- ddpg.py：DDPG 模型的实现。
- main.py：启动的主程序。
- memory.py：Replay Buffer 的实现。
- models.py：Actor 和 Critic 的实现。
- noise.py：随机噪声的实现。
- training.py：训练过程的实现。

我们可以从一些独立的部分入手进行分析，首先是 noise.py，其中介绍了几种噪声生成方法。

- NormalActionNoise：从高斯分布 $N(\mu, \sigma)$ 中采样噪声。
- AdaptiveNoise：自适应方差的噪声，可以根据随机的效果调整采样的方差。
- OrnsteinUhlenbeckNoise：前面提到的噪声实现。

memory.py 中包含 Replay Buffer 的实现，这里实现了最基本的 Replay Buffer。在阅读了前面几章的代码后，我们对这部分的代码已经很熟悉了。

在 models.py 中介绍了两个模型，由于我们解决的问题是连续的状态和行动空间，不像前面介绍的 Atari 游戏那样以画面为状态输入，因此模型只需要全连接层就可以。两个模型都采用 3 层全连接层实现，其中 Actor 模型的输出和行动的维度相同，Critic 模型的输出维度为 1。

ddpg.py 中介绍了训练的主要内容。由于我们采用了回归模型进行模型训练，数值的范围变得不太可控，为了让训练过程变得更稳定、更快捷，需要对输入和输出进行归一化。对于输入的观测值，我们首先经过一个滑动平均的结构记录输入值的均值和方差，然后在输入值进入模型前对其进行归一化，也就是减去它的均值和方差。同样，对于输出值，我们得到的也是被归一化之后的数值，在模型输出后，这个数值再被恢复到原来的数值，这个过程如图 11-4 所示。

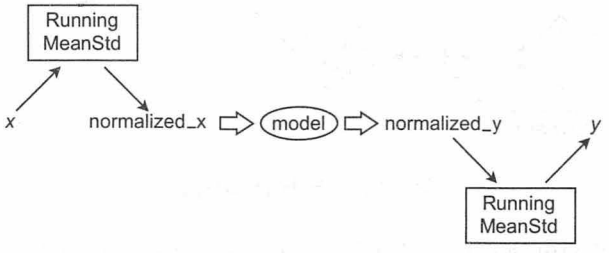


图 11-4　输入和输出的归一化与恢复

模型在训练时需要从 Replay Buffer 中读取数据，我们首先使用 Q-Learning 计算更新后的价值。我们从数据中找到 $t+1$ 时刻的状态 obs1、回报 reward 和状态是否为终结状态的标识符 terminate，使用 Target Network 中的 target_actor 和 target_critic 两个模型就可以计算得到：

```
target_Q = reward + (1-terminate)* gamma * target_critic(obs1,target_actor
(obs1))
```

得到它之后，就可以计算价值模型的目标了。我们再使用 t 时刻的状态 obs0，就可以得到：

```
critic_loss = sqrt(square(target_Q - critic(obs0, action)))
```

我们的目标就是最小化上面的公式。上面的公式里我们忽略了归一化相关的操作，在代码中我们会看到这部分的内容。而对策略模型，我们的目标为：

```
actor_loss = -critic(obs0,actor(obs0))
```

只要对这些目标进行求导就可以了，这部分的计算如图 11-5 和图 11-6 所示。

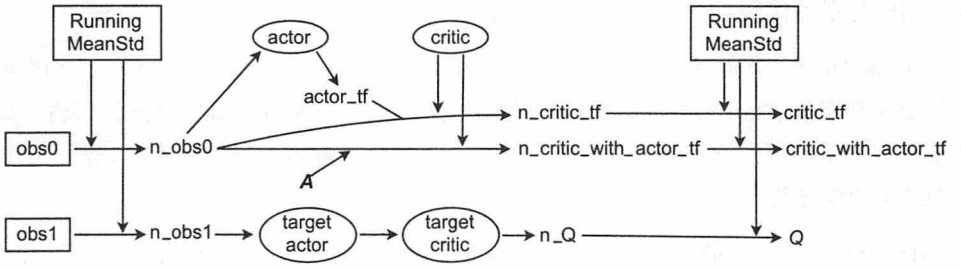


图 11-5　目标计算图 1

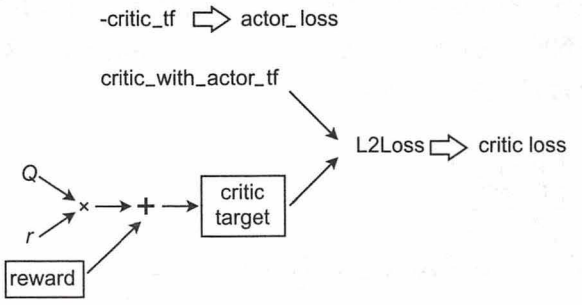


图 11-6　目标计算图 2

代码中使用了 MPI 进行梯度的通信，这样增强了模型的并发度。以上就是代码的核心部分，对具体内容感兴趣的读者可以阅读其中的细节。

11.4 总结

本章我们主要介绍了基于 Off-Policy 的 Actor Critic 算法，在此我们总结如下。

（1）ACER 算法继承了 Off-Policy Actor Critic 的算法，通过对重要性采样比例的限制与偏差纠正，减少了因重要性采样造成的优化波动，同时使用 Retrace 方法减少价值估计的波动。

（2）DDPG 算法吸收了 DQN 的思想，采用确定的策略函数，使问题在高维度的连续空间上能够发挥更好的效果。同时，DDPG 算法吸收了大量 DQN 的改进方案，使模型在训练频率和效果上有了保证。

11.5 参考资料

[1] Munos R, Stepleton T, Harutyunyan A, et al. Safe and Efficient Off-Policy Reinforcement Learning[J]. 2016.

[2] Degris T, White M, Sutton R S. Off-Policy Actor-Critic[J]. 2012.

[3] Wang Z, Bapst V, Heess N, et al. Sample Efficient Actor-Critic with Experience Replay[J]. 2016.

[4] Silver D, Lever G, Heess N, et al. Deterministic policy gradient algorithms[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2014:387-395.

[5] Lillicrap T P, Hunt J J, Pritzel A, et al. Continuous control with deep reinforcement learning[J]. Computer Science, 2015, 8(6):A187.

[6] Večerík M, Hester T, Scholz J, et al. Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards[J]. 2017.

第四部分
其他强化学习算法

第四部分将介绍其他强化学习算法。由于强化学习本身十分复杂，需要解决的问题很多，前面介绍的算法并不能涵盖所有的内容，仍然有很多优秀的算法和有意义的问题值得我们探索。笔者挑选了两个比较经典的子领域介绍给读者，希望能够扩展读者对强化学习认知的广度。

第四部分共包含两章：

- 第 12 章将介绍解决回报稀疏问题的算法，包括层次强化学习（HRL）和 Hindsight 经验回放（HER）。
- 第 13 章将介绍结合状态转移概率的算法，包括棋类博弈常见的蒙特卡罗决策树（MCTS）和 iLQR 算法。

书籍是人类进步的阶梯

12 稀疏回报的求解方法

在前面的章节中曾经介绍过 Atari 游戏环境，它的状态观测值是游戏画面，行动是离散且有限的游戏柄操作。在大家的印象中，电子游戏可以给予玩家丰富的反馈，每当玩家取得一点点成果，游戏环境就会给予玩家一定的回报，因此电子游戏环境是非常适合强化学习的。然而，一些游戏的回报反馈十分缓慢，需要玩家执行一系列很长、很复杂的操作，才能获得明确的回报反馈，而在获得回报之前，玩家很难获得任何反馈。实际上，很多强化学习的任务都存在着反馈稀疏的问题，本章就尝试解决这类问题。

12.1 稀疏回报的困难

第 5 章曾介绍过一款称为蒙特祖玛的复仇（Montezuma's Revenge）的游戏，它属于 Atari 游戏系列中的一款，游戏界面如图 12-1 所示。这是一款 RPG 游戏，玩家通过操纵主角的行动通过一个个房间，找到迷宫的终点并完成游戏。为了通过一些房间，玩家需要操纵角色获取房间内的一些钥匙。当玩家获得了钥匙或者通过一个房间时，才能获得一定的分数，而其他操作不能获得任何分数。

这个游戏的特点是反馈比较稀疏。除了获得钥匙和进入下一个房间这两个情境外，其他的行动都不会获得任何正向反馈。同时主角如果从高空坠落，或者触碰到骷髅头时，将会丢失一条生命。因此，绝大多数的操作都不会获得环境的认可或者否定，如果玩家采取常规的探索手段对环境进行探索，很可能快速丢失所有生命，无法获得任何正向反馈。

书籍是人类进步的阶梯

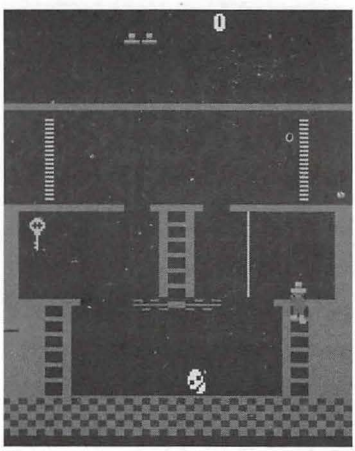


图 12-1　蒙特祖玛的复仇

听到这里，相信读者一定会想：既然触碰骷髅头和从高空坠落这两个操作会丢失生命，那么我们能不能为这两个操作赋予一个负向的反馈呢？这样当然可以，但是这样做可能会出现两个问题：首先在一些场景下，为了通关，我们需要主动牺牲生命来简化游戏，例如，主角触碰骷髅头之后，虽然会损失一条生命，但是骷髅头也会随之消失。因此我们实际上是用一条生命扫清了障碍，这也不失为一种策略；其次，这种方法也被称为**回报重塑**（Reward Shaping），也就是根据任务的具体情况对奖励进行改造。这个方法往往会获得不错的效果，但也需要我们对问题有一定的了解，这样学习到的策略也都与任务相关，并不是通用的策略。这与我们构建通用的人工智能模型的目标有些冲突。

棋类的博弈游戏也存在这样的问题，对围棋来说，自己走出的每一颗棋子并不能得到具体的反馈，局面的形式只能通过自己的经验估计，而按照规则，只有双方结束放子，才会得出胜负结果，所以很可能需要在完成一个很长的轨迹之后得到反馈。和蒙特祖玛的复仇类似，这对学习并不是很友好，我们需要额外的工作确定回报究竟来源于这段时间内的哪些操作，这个问题也被称为 Credit Assignment Problem，即一局棋下完之后，我们要分析究竟哪些招法是妙招，哪些招法是昏招。

可是令人沮丧的是，Credit Assignment Problem 并不是一个很容易解决的问题。即使是前面介绍的那些反馈频繁的问题，也存在着相同的问题，只不过当反馈足够频繁时，这个问题造成的影响并不大。只要反馈足够频繁，每一个行动的优劣都会随着训练样本的增加变得逐渐明确，所以这个问题一直以来没有被我们当成一个很大的问题；但是对于反馈稀缺的任务，行动的价值很难精确地衡量，这个问题也就被放大了。如果我们想学习较好的策略，那么一个直观的想法是解决反馈稀疏的问题：让反馈变得“稠

书籍是人类进步的阶梯

密”起来。只要反馈变稠密，我们就可以使用前面章节介绍的算法求解，并能获得很好的策略模型。

实际上，对于反馈稀疏的任务，还有一个解决方法，就是增强 Agent 的探索能力，前面我们介绍了 $\epsilon-$greedy 的随机策略方法，它可以提高策略的探索能力，但是这种方法有一定的限制，因为它只能在当前状态附近进行无目标的随机探索，对于持续时间比较短的任务来说，足够数量的随机行动会产生“瞎猫碰上死耗子”的效果，一旦通过随机探索遇到了一些有效的回报，模型就会很快地学习到一些有用的信息，获得反馈的频率也会不断提高；但对更复杂的任务来说，单纯的探索也不一定能学习到很好的策略。

笔者曾关注过一个十分小众的游戏：仙剑 98 速通。玩家要在仙剑 98 这款曾经十分有名的游戏上尽可能快速地完成游戏。据笔者所知，完成游戏的最快时间在 142 分钟左右，比起初玩游戏时花费数天才能完成游戏，这个速度让许多人感到惊叹。玩家想要做到这个速度，需要具备以下条件。

（1）通过所有迷宫时都做到不绕路，不执行多余的操作。图 12-2 所示的地图，是游戏中一个很经典、很难的地图，站在“上帝”视角我们也会觉得这个迷宫比较复杂，更别说玩家在游戏时只能看到很小一部分的场景，很容易被地图中一些支路带回起始出发点，因此很多玩家都会在这里迷路。想要快速通过这个迷宫，玩家甚至需要利用地图中留下的 Bug 快速通关。这样复杂的操作难以通过单纯探索实现。

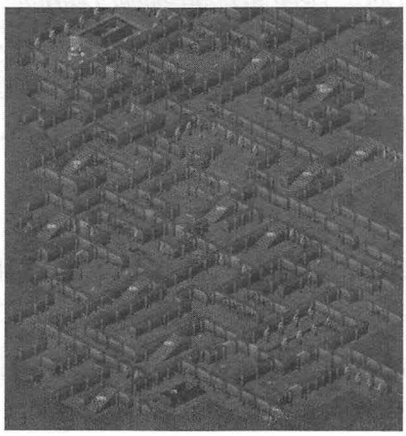


图 12-2 仙剑 98 将军冢二层地图

（2）超长的思考路径。为了获得最快的通关速度，玩家需要在游戏前期获取一些道具，并将这些道具使用到游戏后期。例如，从前面的怪物处偷窃一些攻击性道具，然后投掷到后期的 Boss 身上。或者为了节省游戏时间，缩减一些冗长的剧情，提前购买装备。这些时间跨度非常长的策略也是难以通过单纯的探索得到的。

书籍是人类进步的阶梯

（3）备用策略。游戏中存在一定的随机性，而这是最考验每一个速通者的环节。若没有获得一个减少时间的关键道具，玩家就需要采取其他的策略弥补。弥补的方法很多，玩家需要快速确定一个最优的弥补方案。除此之外，有时玩家还需要根据游戏的形势采取一些冒险的行动，例如放弃存档、放弃补血等。这些都需要经过不断地探索才能掌握。

读者如果对这个游戏有一定的了解，会明白上面提到的三个难点。读者也许会想，既然这个游戏存在这么多难点，人类是如何探索出来的呢？虽然最终的衡量目标是游戏的总时间，人类却可以找到很多游戏内在的小目标，通过这些小目标的辅助，使游戏的探索更高效。强化学习中的 $\epsilon-$greedy 方法显然不能适应这样复杂的游戏环境，如果我们想让机器也获得这样的能力，就需要让它拥有更智能、更理性的探索方法。

通过上面的分析我们发现，解决反馈稀疏的方法主要是通过为 Agent 增加一些内在的目标使反馈变得不再稀疏。本章将介绍两个方法，它们的核心思想来源于上面的两个解决方法。其中一个采用层次强化学习的方法，通过为任务设立一些“小目标”加快学习；另一个则干脆修改了目标，使有效回报的数量变多。

12.2 层次强化学习

本节介绍论文 *Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation*[1] 中介绍的方法：层次强化学习（Hierarchical Reinforcement Learning，简称 HRL），它的目标就是解决像蒙特祖玛的复仇这样反馈稀疏的问题。我们已经知道这个游戏的反馈比较稀少，交互的轨迹比较长，而 HRL 解决问题的方法就是设定一些符合直觉的“小目标”。

12.1 节已经介绍了设立“小目标”（goal）的缘由，那么让我们直接看小目标的设定方法。我们以到达某个状态作为完成这个小目标的条件。那么，整个任务就变成：到达某个状态 1→ 到达某个状态 2→……到达某个状态 N → 完成任务，得到反馈。

在完成一个小目标的过程中，Agent 将不再关注环境本身的反馈，而是专注于完成这个小目标。这样我们的任务就有了层次结构，完成这个任务的 Agent 仿佛不是一个人，而是一个团队。团队的领导者负责制定小目标，而团队的执行者负责完成小目标。经过这样的分解，我们发现一个单独的模型已经无法满足我们的需求，我们需要构建两个模型。我们把团队领导称为元控制器（Meta Controller），它负责接收环境的状态和反馈，并根据这些信息产生小目标。执行人员称为控制器（Controller），它接收环境状态和小目标，并根据这些信息产生行动。两个模型的结构与交互流程如图 12-3 和图 12-4 所示。

书籍是人类进步的阶梯

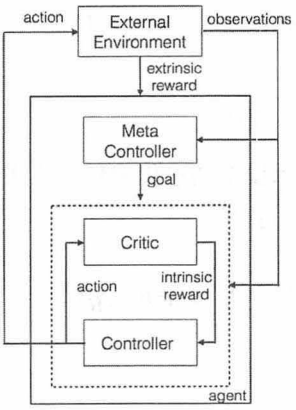


图 12-3 层次强化学习结构图

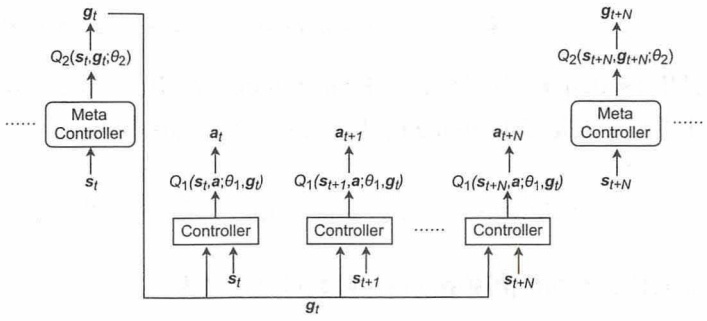


图 12-4 层次强化学习交互图

我们整理出的层次强化学习完整流程分如下 5 步。

（1）环境提供状态 s 和外在反馈 f。

（2）元控制器 π_g 得到内部目标 g，并将状态和内部目标 g 发送给控制器。

（3）控制器 π_{ag} 接收到信息，产生真实的行动 a。

（4）环境接收行动 a，并产生新的状态 s 和外在反馈 f。

（5）元控制器根据新的状态判断内部目标 g 是否完成，如果完成，就生成了一个新的内部目标 g；同时根据目标完成情况给出内部反馈 r，并回到第 3 步。

从时间维度看，控制器在每一个时刻都要根据状态和目标给出行动；而元控制器只在一段时间内执行一次行动。这样两个模型在时间维度上具有一定的层次性。论文中给出了基于 Q-Learning 的学习方法，对控制器来说，它的目标函数为

$$\begin{aligned} Q_1^*(s,a;g) &= \max_{\pi_{ag}} E[\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} | s_t = s, a_t = a, g_t = g, \pi_{ag}] \\ &= \max_{\pi_{ag}} E[r_t + \gamma \max_{a_{t+1}} Q_1^*(s_{t+1}, a_{t+1}; g) | s_t = s, a_t = a, g_t = g, \pi_{ag}] \end{aligned}$$

对元控制器来说，它的目标为

$$Q_2^*(s,g) = \max_{\pi_g} E[\sum_{t'=t}^{t+N} f_{t'} + \gamma \max_{g'} Q_2^*(s_{t+N}, g') | s_t = s, g_t = g, \pi_g]$$

其中 N 表示执行小目标经历的时间。根据第 7 章和第 8 章的介绍，我们可以使用两个模型分别近似这两个控制器的价值估计结果。除此之外，DQN 中的所有训练技巧都可

书籍是人类进步的阶梯

以运用在其中，例如 Replay Buffer、Target Network 等。控制器的价值模型的目标函数可以定义为

$$L_1(\theta_{1,i}) = E_{(\boldsymbol{s},\boldsymbol{a},\boldsymbol{g},\boldsymbol{r},\boldsymbol{s}')\sim D_1}[(y_{1,i} - Q_1(\boldsymbol{s},\boldsymbol{a};\theta_{1,i},\boldsymbol{g}))^2]$$

其中 $(\boldsymbol{s},\boldsymbol{a},\boldsymbol{g},\boldsymbol{r},\boldsymbol{s}')$ 是保存在 Replay Buffer D_1 中的样本，i 表示模型的迭代编号。公式中的 $y_{1,i}$ 就是 Q-Learning 的计算结果。对目标函数求导，可以得到

$$\nabla_{\theta_{1,i}} L_1(\theta_{1,i}) = E_{(\boldsymbol{s},\boldsymbol{a},\boldsymbol{g},\boldsymbol{r},\boldsymbol{s}')\sim D_1}[(y_{1,i} - Q_1(\boldsymbol{s},\boldsymbol{a};\theta_{1,i},\boldsymbol{g}))\nabla_{\theta_{1,i}} Q_1(\boldsymbol{s},\boldsymbol{a};\theta_{1,i},\boldsymbol{g})]$$

而元控制器的价值模型的目标函数可以定义为

$$L_2(\theta_{2,i}) = E_{(\boldsymbol{s},\boldsymbol{g},\boldsymbol{f},\boldsymbol{s}')\sim D_2}[(y_{2,i} - Q_2(\boldsymbol{s},\boldsymbol{g};\theta_{2,i}))^2]$$

其中 $(\boldsymbol{s},\boldsymbol{g},\boldsymbol{f},\boldsymbol{s}')$ 是保存在 Replay Buffer D_2 中的样本。对目标函数求导，可以得到

$$\nabla_{\theta_{2,i}} L_2(\theta_{2,i}) = E_{(\boldsymbol{s},\boldsymbol{g},\boldsymbol{f},\boldsymbol{s}')\sim D_2}[(y_{2,i} - Q_2(\boldsymbol{s},\boldsymbol{g};\theta_{2,i}))\nabla_{\theta_{2,i}} Q_2(\boldsymbol{s},\boldsymbol{g};\theta_{2,i})]$$

以下是层次强化学习的算法。

算法　层次强化学习算法

Input

元控制器和控制器两个模型需要的 Replay Buffer$\{D_1, D_2\}$，以及模型的参数 $\{\theta_1, \theta_2\}$

控制器探索概率 $\epsilon_{1,\boldsymbol{g}} = 1$，元控制器探索概率 $\epsilon_2 = 1$

Start

for $i = 1$, num_episodes **do**

　获取游戏初始状态 $\boldsymbol{s}$

　由 $\boldsymbol{s}$、Q_{θ_2} 和 ϵ_2 得到目标 $\boldsymbol{g}$

　while $\boldsymbol{s}$ 不是终止状态 **do**

　　$\boldsymbol{F} \leftarrow 0$

　　$\boldsymbol{s}_0 \leftarrow \boldsymbol{s}$

　　while $\boldsymbol{s}$ 不是终结状态 and 目标 $\boldsymbol{g}$ 未完成 **do**

　　由 $\boldsymbol{s}$、$\boldsymbol{g}$、Q_{θ_1} 和 $\epsilon_{1,\boldsymbol{g}}$ 得到行动 $\boldsymbol{a}$

　　执行行动 $\boldsymbol{a}$，并从环境获得下一个状态 $\boldsymbol{s}'$ 和外部回报 $\boldsymbol{f}$

　　从内部的评判模型得到内部回报 $r(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')$

　　将样本 $(\{\boldsymbol{s},\boldsymbol{g}\},\boldsymbol{a},\boldsymbol{r},\{\boldsymbol{s}',\boldsymbol{g}\})$ 存入 D_1 中

　　使用 D_1 的样本更新模型参数 θ_1，目标函数为 $L_1(\theta_1)$

书籍是人类进步的阶梯

使用 D_2 的样本更新模型参数 θ_2，目标函数为 $L_2(\theta_2)$

$\boldsymbol{F} \leftarrow \boldsymbol{F} + \boldsymbol{f}$

$\boldsymbol{s} \leftarrow \boldsymbol{s}'$

end while

将样本 $(\boldsymbol{s}_0, \boldsymbol{g}, \boldsymbol{F}, \boldsymbol{s}')$ 存入 D_2 中

if $\boldsymbol{s}$ 不是终结状态 **then**

由 $\boldsymbol{s}$、Q_{θ_2} 和 ϵ_2 得到目标 $\boldsymbol{g}$

end if

end while

衰减 ϵ_2，并按照目标 $\boldsymbol{g}$ 的完成情况衰减 ϵ_1

end for

笔者以论文中的一个例子为例，介绍层次强化学习的实际应用。这个例子的环境如图 12-5 所示。图中共有 6 个状态，从左到右依次为 $\boldsymbol{S}_1$ 到 $\boldsymbol{S}_6$，Agent 共有两个操作，向左移动和向右移动。当 Agent 决定向左移动时，它会移动到左边的状态；当 Agent 决定向右移动时，它有 50% 的概率移动到右边的状态，也有 50% 的概率移动到左边的状态。任务的终结状态是 $\boldsymbol{S}_1$，在任务过程中，我们不会得到任何反馈，直到任务终止，才会根据任务完成情况获得反馈。最终奖励和 Agent 是否经过某个状态有关，如果 Agent 首先经过 $\boldsymbol{S}_6$ 才到达 $\boldsymbol{S}_1$，那么 Agent 将获得 1 单位奖励；如果没有经过则只会获得 0.01 单位的奖励。

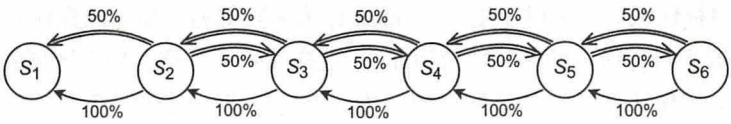


图 12-5 随机过程环境

为了最大化奖励，我们当然希望 Agent 首先来到环境的最右边，然后回到左边，但由于 Agent 在交互时并不知道这样的规则，无法从中总结出规律，因此直接进行交互时，它的行为会显得有些盲目。如果将这个任务改造成层次强化学习的问题，就会变得完全不一样。我们让元控制器先从 6 个状态中选择一个状态作为目标点，于是当 Agent 到达这个状态时，它会获得一份奖励。这样虽然我们不明确指出真正有价值的状态，Agent 还是可以通过小目标的建立更快速地找到其中的规律。

除了上面这个小例子，我们回到本章开始处介绍的 Atari 游戏：蒙特祖玛的复仇。采用层次强化学习法，问题将被转换成一个层次的问题，元控制器首先负责定义一个状态作为小目标，控制器通过探索与交互完成目标。只不过这个问题不同于前面的例

书籍是人类进步的阶梯

子，我们无法用抽象的形式表示状态，而且图像中的可行状态有很多，将所有的状态找出来也不太可能。为此，我们采用另外一种办法，利用图像自身的特点，从图像中利用无监督的方法找到一些显著的物体，例如图像中的钥匙、梯子、骷髅头等。我们可以使用很多图像处理的方法找到这些实体（例如，基于图像分割的方法、实体检测的方法、显著物体检测的方法等），找到了这些实体，再使用元控制器决定当前的小目标是什么。选择小目标的效果如图 12-6 所示。

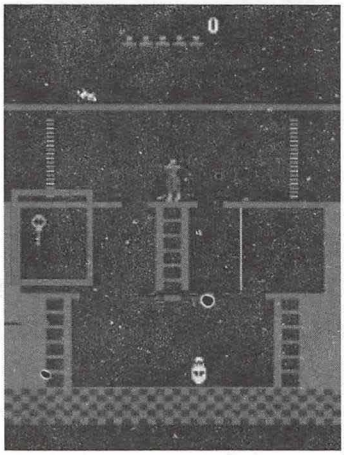


图 12-6　选择小目标的效果

一旦确定了小目标，控制器就可以操纵 Agent 完成小目标。我们使用元控制器选定了一个小目标区域后，可以使用一个 mask 作为单独的通道加入图像输入中，在这个 mask 中，只有小目标所在的区域数值为非 0，这样就强调了这个局部区域的重要性。我们将组合后的图像传入卷积神经网络模型中，就可以得到行动的概率分布，并使用概率分布采样得到行动。对应的目标函数也比较好建立，这里不再赘述。

我们通过使用层次强化学习，使任务有了小目标，这样 Agent 就可以更容易地找到一些探索方向，从而加快解决问题的速度。对一些回报稀疏的问题来说，这样的方法具有很好的效果。

12.3　HER

7.8 节和 8.2 节分别介绍了 Replay Buffer、Priority Replay Buffer 两个存储训练样本的数据结构。它们都具有减少序列样本相关性、提高样本利用率的功能，同时在算法中被广泛应用。本节介绍另外一个存储样本的结构 Hindsight Experience Replay（简称

书籍是人类进步的阶梯

HER），来自同名论文[2]。它的主体结构和前面介绍的两种结构基本一致，主要的不同之处在于它对样本回报和价值的设定。HER 可以用在回报稀疏的环境中，为 Agent 提供更多的回报信息。

12.3.1 渐进式学习

前面已经提到了一些反馈稀疏的任务，为了介绍 HER 这个新结构，论文介绍了一个十分特殊的任务。在任务中，环境会初始化产生两个长度为 N，由 0 和 1 组成的数组，其中一个为初始值，另一个为目标值。Agent 在每一时刻可以翻转任意一位数字，将其从 0 变成 1，或者从 1 变成 0。当 Agent 操纵的数组和目标数组完全一致时，Agent 就会获得奖励；其他情况下，Agent 不会获得任何奖励。任务的时间长度有一定的限制，到达指定的时间后任务结束。这个任务的流程如图 12-7 所示。



图 12-7 数字翻转任务示意图

可以看出，任务的反馈十分稀疏，而且数组的长度越长，反馈越稀疏。如果策略不够优秀，Agent 无法将数组移动为目标值，也无法获得任何反馈。如果持续保持这样的学习状态，Agent 需要很长时间才能学习到最优策略，或者无法学习到。

既然问题出自反馈，那么能不能增加反馈出现的频率呢？假设初始状态为 s_0，目标状态为 s_t，经过 T 轮的行动操作，得到状态 s_T，虽然现在的方法没有办法在 T 时刻内将 s_0 变换到 s_t，但是可以变换到 s_T。所以我们不应该抱怨模型的效果不好，没有让我们完成既定目标，而应该埋怨目标设置得不合理，如果把目标换成 s_T，策略模型不就完成任务了吗？

读者一定会对这样的思想产生质疑，我们的目标明明是 s_t，为什么要把它修改到 s_T 呢？这种行为相当于纵容我们的模型，对它错误的表现提出了鼓励？这就好像学生参加考试，考试的满分是 100 分，通过分数是 60 分。一位同学经过几个小时的努力，终于考出了 50 分的“好成绩”，于是大家纷纷抨击考试制度：明明 50 分已经很优秀了，

书籍是人类进步的阶梯

为什么要将门槛设置为 60 分？你知道这位同学多努力学习吗？他在发烧的状态下还坚持每天学习 25 个小时……但是在读者的眼中，成绩没考过 60 分，就是没有通过，任何解释都没有意义。这样的想法并不奇怪。

如果这个考试是一个正常的考试，维持标准无可厚非。如果有些同学的考试分数超过了 60 分，那么至少证明达到通过的分数是可行的，那些没达到分数的同学也没什么好抱怨的。但是如果给一群小学生出了高等数学题目，让小学生使用泰勒公式、欧拉定理和勒贝格积分，估计家长会说老师教学方式有问题。如果老师真的出了这样一张卷子给小学生，那么小学生答多少分都情有可原，就算不到 60 分也没关系；如果小学生竟然蒙对了一些题目，说不定家长还会觉得小学生具有数学天分。

所以到这里，我们发现了问题的关键：卷子的难度和考生的能力是否匹配。如果基本匹配，那么常规的标准设定是合情理的；如果完全不匹配，很多设定就无法按照常理进行。我们知道小学生几乎无法解决高等数学问题，所以 60 分的通过线是不合理的。同理，对一个几乎没有反馈的强化学习任务来说，“通过反馈影响决策”这个机制根本无法建立，我们也不能保证模型能学到多少有用的东西。既然这样，我们为什么还要要求模型一定要达到原本规定的目标呢？我们能不能降低一点难度呢？

机器学习的研究问题中有一个与上面提到的问题相关的领域，叫作**课程学习**（Curriculum Learning）。Curriculum 的含义是“课程”，而 Curriculum Learning 在机器学习中代表一种渐进式地学习策略。这种学习思想同样来源于人类自身，我们不可能让一个刚出生的婴儿直接参加高考并取得优异的成绩，因为这对婴儿来说太难了。直接挑战这样的任务不但十分困难，也更容易在学习的过程中学习到“投机取巧”的解法，例如大家熟知的“三长一短选最短，三短一长选最长”、“不会就蒙 C”等。这些基于“大数据”分析得到的经验在一些“数据集（考试卷）”上会有良好的效果，但是它实际上是一种过拟合的表现，一旦出题人改变出题思路，结果将变得很糟糕。所以一般来说，我们要采取渐进式的学习方法，在学习的道路上多设置一些“小目标”，让自己在不断完成小目标的过程中成长。

那么这样的小目标该如何设定呢？一般来说，我们可以从几个方向实现。以打靶为例，一个人手持气枪瞄准一个靶子射击，靶子上有一些标识，给不同的区域设定了各自的分数，同时人和靶子要保持一定的距离，射击还包括时间和其他因素的限制。如果训练一个射击选手，那么我们可以使用以下几个思路进行训练。

（1）任务分解。我们将任务分解成一些更小粒度的任务，例如先练习持枪，如何稳定地握住枪而不发抖；其次练习瞄准，可以在枪的前方安装一个激光器，让选手练习把激光射向想要的分数区域；最后让选手适应射击时的后坐力，将后坐力造成的震动考虑到射击的过程中。通过完成这样一个个的小任务，大任务可以完成得更好。实际上 12.1 节介绍的方法和上面介绍的方法更相近。

书籍是人类进步的阶梯

（2）调整难度。一开始我们缩短人和靶子的距离，同时扩大高分的区域，让选手在一开始也可以获得一定的分数，这样可以使选手建立最初步的射击策略；选手的水平又提高后，将距离变长，同时减少高分区域，进一步提高选手的水平。渐渐地，选手的能力不断提高，而选手的射击环境和真实的环境一致，选手完成了渐进式地学习。这种方法就是课程学习的主要思想。

介绍了这么多关于课程学习的内容，让我们回到前面的例子。我们把最终目标变为 s_T，实际上相当于降低了任务的难度。我们可以**通过调整任务难度让模型渐进式地学习，不断增强策略的能力**。这就是 HER 背后的思想。Hindsight 可以翻译为“后见之明”。读者一定听说过“先见之明”，先见之明（Foresight）是指在遇见一个事物之前就对它有准确判断，而后见之明是指遇到一个事物之后的准确判断。论文中也提到，人类与模型相比有一个明显的长处，那就是在无法得到任何明确的反馈时，依然可以从中得到一定的收获并积累经验；而计算机就不能获得任何的收获。人类的这个特点被称为“后见之明”，用俗语来说就是“事后诸葛亮”。

我们可以用“后见之明”的思想重新描述 HER 方法：当我们经过操纵得到状态 s_T 时，虽然我们没有得到反馈，但通过操作我们可以得到状态 s_T，这个经验还是可以保留下来的，万一下一次目标状态就是 s_T 呢？

12.3.2 HER 的实现

介绍完 HER 的基本思想，下面来看看 HER 的具体实现，代码在 baselines/her/中，项目通过使用 HER 实现了 DDPG 算法。关于 DDPG 的实现已经在 11.3 节介绍过，这里不再赘述。我们将目光集中在数据收集和采样这两个环节，其中涉及三个代码文件的内容。

首先是 Replay Buffer 的数据结构，代码在 replay_buffer.py 中。代码中构建的数据结构如图 12-8 所示。

采样数据存在一个字典（dict）中，Replay Buffer 存储了如下一些 key。

o：代表观测值（observation）。

u：代表行动值（action）。

g：代表目标值（goal）。由于算法可以通过目标计算得到回报，因此这里将不再存储回报值。

ag：代表已经完成的目标（achieved goal），也就是交互终止时的状态值。

除了以上 4 个信息，在采样时，Replay Buffer 还将生成另外两个 key。

o_2：代表下一时刻的观测值。

ag_2：代表下一时刻已完成的目标值。

为了更好地实现 HER 算法，Replay Buffer 以序列（episode）为单位进行存储，也就是说，将一整条序列完整地存储，序列内每一时刻的状态和行动也会如交互时一样排列好。那么，Replay Buffer 可存储的样本容量为 $N \times T$，其中 N 表示 Buffer 容纳的序列数量，T 表示每一个序列的长度，其他的结构与前面介绍的 Replay Buffer 类似。

在采样交互的过程中，策略模型同环境交互，将得到一个完整的交互序列 $\{\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{g}_0, \ldots, \boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{g}_t\}$。由于每一次交互我们都会得到下一时刻的状态，所以可以将其保存为前一时刻的 achieved goal，意思为已经达到的“目标”。这样序列就变成了 $\{\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{g}_0, \boldsymbol{ag}_0, \ldots, \boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{g}_t, \boldsymbol{ag}_t\}$，我们将这个序列完整地保存到 Replay Buffer 中。这部分内容在代码 rollout.py 中可以看到，这部分过程如图 12-9 所示。

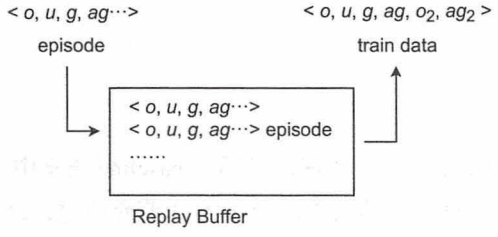


图 12-8　Hindsight Replay Buffer 的结构图

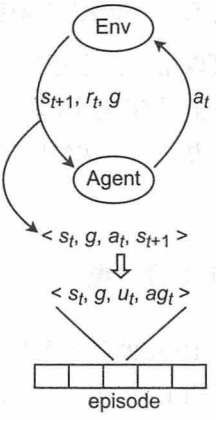


图 12-9　样本采样与处理

在训练的过程中，我们需要对从 Replay Buffer 中采样出的样本，设定一定的概率将它的目标从原定的目标 $\boldsymbol{g}$ 变成其他容易实现的目标。这里我们采用论文中提到的 future 采样模式，假设采样得出样本的时刻为 i，那么我们将从 $j \in [i+1, T]$ 的时间区间选择一个时刻的 achieved goal 作为样本的目标，也就是说：

$$\boldsymbol{g}_i \leftarrow \boldsymbol{ag}_j, j \in [i+1, T]$$

那么，完整的样本选取过程将会变成下面的流程。

（1）从 Replay Buffer 的 N 个序列中采样 b 个序列。

（2）从 b 个序列中各选出一个时刻的样本，得到 b 个样本。

（3）每一个样本有一定的概率将 achived_goal 设置为当前时刻后的任一时刻的状态。

样本选取的过程如图 12-10 所示。

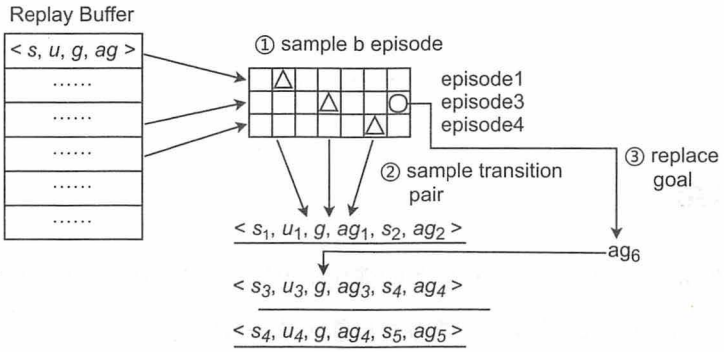


图 12-10 选取训练样本流程

这样我们就了解了 HER 的全部内容。HER 使我们从课程学习的角度解决了反馈稀疏的问题，并获得了一定的成功。最后给出 HER 算法的全过程伪代码：

算法 Hindsight Experience Replay 算法（基于 future 采样方法）

Input

off-policy RL 算法 A：例如 DQN（7.9 节介绍）、DDPG（11.2 节介绍）

回报函数 r：$\boldsymbol{S} \times \boldsymbol{A} \times \boldsymbol{G} \rightarrow \boldsymbol{R}$

Replay Buffer：$\boldsymbol{R}$

Start

初始化 $\boldsymbol{A}$，初始化 $\boldsymbol{R}$

for episode $= 1$，M **do**

采样目标 $\boldsymbol{g}$ 和初始状态 $\boldsymbol{s}_0$

for $t = 0, T-1$ **do**

从算法 A 中的策略 π_b 得到行动 $\boldsymbol{a}_t$：$\boldsymbol{a}_t \leftarrow \pi_b(\boldsymbol{s}_t \| \boldsymbol{g})$

执行行动 $\boldsymbol{a}_t$ 并获得新的状态 $\boldsymbol{s}_{t+1}$

end for

for $t = 0, T-1$ **do**

$\boldsymbol{r}_t := r(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{g})$

将样本 $(\boldsymbol{s}_t \| \boldsymbol{g}, \boldsymbol{a}_t, \boldsymbol{r}_t, \boldsymbol{s}_{t+1} \| \boldsymbol{g})$ 保存到 $\boldsymbol{R}$ 中

end for

for $t = 1, N$ **do**
从 $\boldsymbol{R}$ 中采样一个 minibatch B，每个样本有一定的概率将 $\boldsymbol{g}_t$ 修改为 $\{\boldsymbol{s}_{t+1}, \ldots, \boldsymbol{s}_{T-1}\}$ 中的任意一个元素
使用 minibatch B 对 $\boldsymbol{A}$ 进行优化训练
end for
end for

12.4 总结

本章我们介绍了解决反馈稀疏任务的强化学习算法，让我们总结一下。

（1）当反馈比较稀疏时，强化学习有可能无法完成学习。

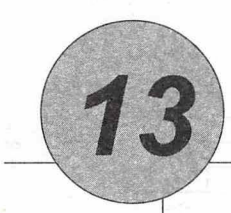
（2）HRL 方法通过构建层次式的决策模型和设立“小目标”的形式增加了环境给 Agent 的反馈。

（3）HER 通过设立相对可行的目标，使 Agent 的学习难度变得相对平滑。

12.5 参考资料

[1] Kulkarni T D, Narasimhan K R, Saeedi A, et al. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation[J]. 2016.

[2] Andrychowicz M, Wolski F, Ray A, et al. Hindsight Experience Replay[J]. 2017.



Model-based 方法

第 6 章介绍了强化学习最基本的概念，那时我们假设自己处于上帝视角，对环境的运转情况完全了解，这样我们就可以采用动态规划的方法，使用表格式的建模形式解决问题，使用的算法有策略迭代法、价值迭代法和广义策略迭代法。后来我们的视角就转移到了环境未知的问题上，为了解决这些问题，我们引出了一系列的方法，如 Q-Learning 和 Policy Gradient。

我们发现前面的很多方法在面对环境未知的问题时采用了比较“被动”的方法：既然环境的状态转移是未知的，那么我们就不去关注它。本章我们要采取更积极的方法，也就是 Model-based 的方法，它通过对状态转移概率进行建模，使我们对强化学习的整个流程进行整体优化。本章先介绍影响全世界的 Alpha Zero 模型，其次介绍连续状态行动空间下的经典算法 LQR。

13.1 AlphaZero

13.1.1 围棋游戏

AlphaZero 是一个世人皆知的人工智能模型，经过几个版本的迭代，模型不断接近完美，从最初的以监督学习为核心的训练方法，到使用强化学习为唯一训练的方法，AlphaZero 完成了人工智能的革命，也将人工智能的棋类游戏带入了一个新的高度，相信读者对这段传奇故事已经有所了解。

在介绍 AlphaZero 模型之前，让我们先看看围棋游戏的定义。围棋游戏是一个信息完全公开的零和博弈。参与游戏的主体有两方，如果把一方看作 Agent，那么棋局和另一方可以看作环境。在棋局中，双方交替在棋盘上落子，当双方无法落子时，游戏会按照规则判定最终的获胜方是谁。如果 Agent 胜利，则获得 1 分回报，反之则获得 −1 分，这个过程如图 13-1 所示。

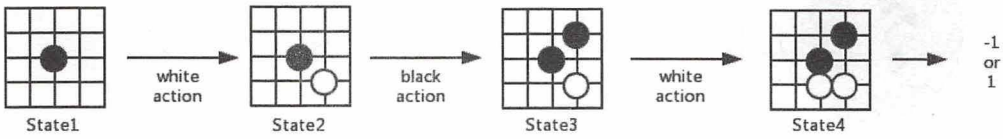


图 13-1　围棋游戏示意图

虽然我们知道对手所有可能的行动，但并不知道对手真正会走到哪个位置。因此，我们可以把对手的策略想象成环境的状态转移概率，这样围棋的问题形式就和前面的问题很相近了。因为游戏的目标是获得最后的胜利，所以我们的目标可以写作

$$\begin{aligned} J(\theta) &= E_{\pi_\theta}[V(\boldsymbol{s}_0)] \\ &= E_{\pi_\theta}[\sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}_0) \sum_{\boldsymbol{s}_1} p(\boldsymbol{s}_1|\boldsymbol{s}_0, \boldsymbol{a})[\boldsymbol{r}_1 + \gamma V(\boldsymbol{s}_1)]] \end{aligned}$$

理论上我们可以采用前面章节介绍的方法进行学习训练，但是这样的方法很难学习到很好的策略模型，因为游戏中的对手可能是一个水平很高的选手，一旦玩家发生一点点失误，对手都可能抓住机会赢得胜利。因此，我们需要在前面算法的基础上进行扩展，获得更多对棋局有利的信息。

我们常说“下棋看三步”，即在落子之前，玩家要考虑落子之后对手的出招，然后考虑应对对手的措施。这种经过深思熟虑的招法往往攻守兼备，招法具有一定威力的同时也避免给对手留下机会。棋类游戏的高手基本上都使用了这种思维方式，他们可能会向后思考四步、五步，甚至更多步。

那么玩家该如何向后思考更多步呢？此时玩家并不知道对手的情况，实际上很难对对手的招法进行估计。那么有没有什么方法解决这个问题呢？对于棋类高手来说，他们一般会把自己当作对手，以自己的棋力估计对手的招法。这个方法虽然不一定能猜中对手的真实招法，但确实是目前最好的方法。如果我们能够运用自己的棋力模拟完成双方的对弈，就可以更准确地估计招法对棋局的效果。这样的方法被称为自我对弈（英文为 Self-Play），即自己和自己下棋。

基于这样的假设，我们可以认为围棋这个游戏在玩家的眼中是一个模型已知的问题。玩家不再将对手想象成环境，因为此时对手也是玩家本身，只不过在不同的时刻，

玩家需要完成不同的目标。玩家在奇数时刻要完成的目标，和在偶数时刻要完成的目标相反。玩家要通过不断地切换自己的身份估计局面的走向，并判断每一个行动最终可能的结果。这个过程如图 13-2 所示，从当前状态出发，所有的棋局分析可以形成一棵树。

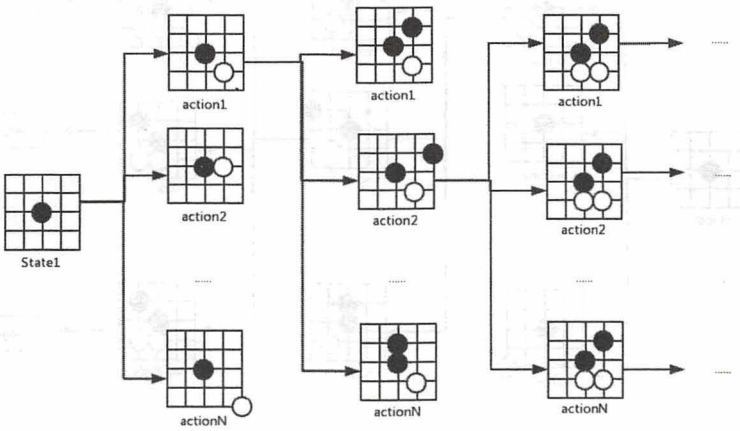


图 13-2　局面与策略估计

13.1.2　Alpha-Beta 树

从图 13-2 中可以看出，为了知道招法的真实效果，我们可以将当前局面下所有的情况都列出来，也就是说先把当前局面下的所有招法列出来，然后把下一轮对手的所有招法列出来，依此类推直到棋局结束，但是这样做的代价实在非常大，这个计算量是计算机不可接受的。既然列出所有的招法不可能，那能不能只列出有限的招法呢？虽然棋盘上有很多的位置可以落子，但有价值的招法应该是有限的。既然我们拥有策略模型，就可以根据策略计算出有价值的前 N 个招法，在树的宽度上进行了裁剪。这样计算量在一定程度上减少了。

除此之外，我们还希望能在树的深度上做削减。如果对每一个棋局使用策略模型计算接下来的招法，那么当树的层数为 d，每一层展开 N 个局面时，我们仍然需要计算这么多的局面：

$$= 1 + N + N^2 + \cdots + N^{d-1} = \frac{N^d - 1}{N - 1}$$

可以看出，策略的使用次数随着树的深度呈指数形式增长，因此控制树的深度是必需的。这时我们想到了价值模型，可不可以采用一种折中的方法，当使用策略计算到第 d 层时，即使我们没法确定局面的最终结果，也可以停止进一步训练，使用一个价值

模型判断双方的局势，并估计该局面下最终的期望结果呢？这相当于我们放弃了一定的精度来换取时间，对应的思路如图 13-3 所示。

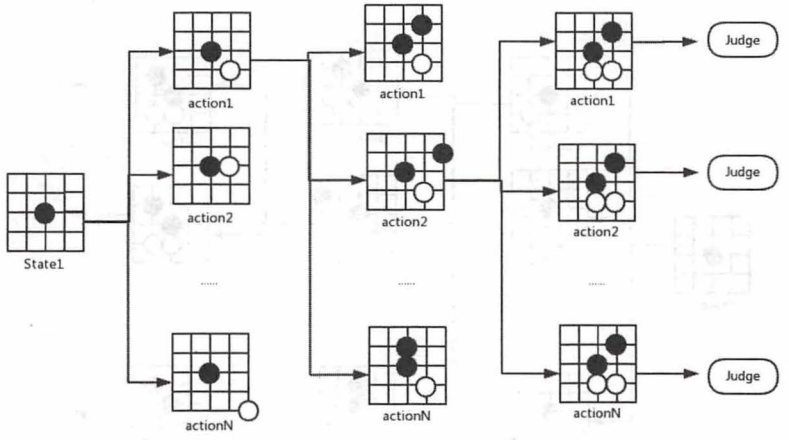


图 13-3　包含价值模型的树

通过上面的两种方法，我们可以很好地减少棋局计算的数量，还可以通过调节树的宽度和深度调整策略计算的效果。增加宽度相当于增加策略的探索能力，可以发掘到更多的招法；增加深度相当于降低价值模型估计的难度，使我们对最终局面的估计更准确。以上思路通常被称为 Alpha-Beta 搜索法，其中需要策略模型和价值模型的合作，对应的伪代码如下所示：

```
def alpha_beta(board, depth, min, max):
    if depth > N:
        return evaluation(board)
    moves=choose_n_move(board)
    for move in move:
        do_move(board)
        score = -alpha_beta(board, depth+1, -max, -min)
        undo_move(board)
        min = max(min, score)
        if min >= max:
            break
    return min
```

代码中的 min 和 max 代表搜索到当前节点时对局面估计的得分范围。如果我们认

为最终胜利为 1 分，失败为 -1 分，那么初始局面下 $\min = -1$，$\max = 1$。随着搜索与计算的不断深入，这个范围会不断缩小，min 和 max 最终变成一个相同的数值。代码中的 choose_n_move 方法表示使用策略模型对局面进行分析，得出概率最大的前 n 个行动，我们可以通过计算 $p(\boldsymbol{a}|\boldsymbol{s})$ 的结果进行比较；代码中的 evaluation 方法表示使用价值模型对局面进行估计，得到对局面期望的估计。由于我们需要对树进行搜索，因此方法采用了递归的形式，读者需要注意：每当算法要进入下一层迭代时，对结果的估计需要反过来，因为围棋是一个零和博弈游戏，一方的优势就是另一方的劣势。希望读者能够理解这个算法的精髓。

13.1.3 MCTS

前面介绍了 Alpha-Beta 树的基本形式，这个方法在博弈问题中被广泛使用，也获得了不错的效果，而本节介绍的蒙特卡罗树搜索方法，在实际中更具有灵活性。它的核心搜索过程主要分为以下四步。

（1）选择（Selection）：从已知的树节点中选出一个最有“价值”的节点，这个价值由节点的“重要性”决定。

（2）扩展（Expand）：在第一步选择的节点上随机扩展一个或者更多的节点。

（3）仿真（Simulation）：从第二步扩展的节点中选择一个节点，采用某种策略随机生成新的状态，并在这个新的状态上继续利用这种 policy 生成新的状态，这样不断进行下去直到最终状态。

（4）回传（Back-propagation）：将最终的计算结果反向传导给上层的所有父节点，更新父节点的相应信息。

其伪代码和流程如下所示：

```
function MCTSSEARCH(s_0)
    创建根节点 v_0，其中包含状态 s_0
    while 有时间和资源继续计算 do
        v_l ← TREEPOLICY(v_0)
        Δ ←DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0))
```

对应的流程图如图 13-4 所示。

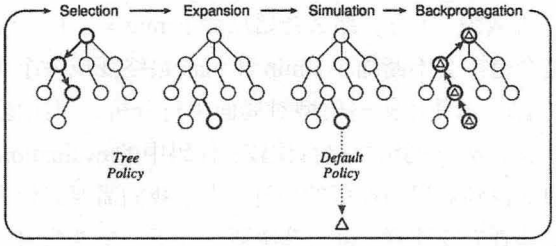


图 13-4　MCTS 的计算步骤

下面笔者详细介绍这四个步骤。

（1）选择。在每次进行蒙特卡罗演绎前，我们需要选定一个树中的节点作为蒙特卡罗过程的出发点。这个过程十分重要，而且蒙特卡罗树搜索的过程和强化学习中的经典问题——探索-利用问题很相似。

前面的章节中曾介绍过，**探索**是尝试之前没有尝试过的行动，如果能找到一些更好的行动，最终的目标也可以完成得更好。我们希望能多尝试一些不同招式的棋局和之前判定胜率低的棋局，因为对手有可能走出我们没有想到的招式，而之前判定为胜率低的棋局也可能是随机采样的次数不够导致的，一旦采样达到一定数量，它就能够真实反映招式的效果，那时也许我们会发现它其实是一个胜率很高的招式。**利用**是指利用当前已知的情况实施策略。对 MCTS 来说，如果已知一个招式的胜率很高，为了进一步明确这个招式真的胜率很高，就需要把有限的采样次数尽可能地用在这种局面上，使我们更确定这个招式的胜率。

前面章节中已经介绍，探索和利用似乎是一对矛盾体，当总资源一定时，将更多的资源分配给当前表现优秀的招式，就会忽略一些潜力股，反之亦然。这就是强化学习中提到的探索-利用困境。只要资源有限，就无法摆脱这个困境。经过大量的实验，人们总结出了一套启发式的策略：在学习过程的初期，策略将向探索方向倾斜，而到了学习过程的后期，利用将占据主导地位。

为了平衡探索与利用，我们还需要将树中每一个节点的“重要性”量化。研究人员发明了很多基于 MCTS 的算法，其中比较经典的算法是 Upper Confidence Bound for Trees 算法，简称 UCT，其中计算节点权值的公式 UCB1 为

$$\mathrm{UCT} = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

其中 $\bar{X}_j$ 表示这个节点当前所有蒙特卡罗过程后的平均得分，这象征了这个节点的招法经过蒙特卡洛演绎后的效果；C_p 是一个参数，用于调整模型在探索和利用之间的权重关系；n 为当前已经进行的蒙特卡罗的轮数；n_j 表示该节点已经进行的蒙特卡罗的轮

数。从这个公式可以看出，想让一个节点被选中，要么是因为它最终获胜的概率高，也就是公式右边的第一项大；要么是它被选中的次数少，这样公式右边的第二项就会大些。这个公式并不复杂，它在实际应用中的效果也很不错，所以被广泛采用。选择过程的目标是从根节点出发找到第一个没有被完全扩展的节点，所以真正的选择过程可能要持续很多轮。

（2）扩展过程。找到符合条件的节点后，就可以对节点进行扩展。扩展实际上是和选择过程结合执行的，选定好待扩展的节点后，算法会从这个节点选择一个没被执行过的行动，执行这个行动并得到下一个状态，然后根据这个状态创建节点，这样树的扩展就算完成了。

（3）模拟过程。这个过程出现在很多棋类博弈算法及初代 AlphaGo 算法中，它会使用一个相对简单的策略模型快速执行双方的招式，每一时刻只从策略中随机选取一个招式并执行，就这样不断地模拟双方的招式直到游戏结束，并记录最终的结果。一般来说，为了保证双方的出招不至于太过离谱，策略需要有一定的棋力，但基于时间的限制，模型又不能过于复杂。

（4）回传过程。当新节点完成模拟后，无论我们采用蒙特卡罗法还是价值估计法，都会得到当前状态的一个结果估计值。由于这个棋局是其父节点可能产生的局面之一，因此可以把得到的结果回传给前面所有的父节点，并更新它们的估计值。

以上就是 MCTS 算法的主要流程。下面我们以 UCT 为例，介绍它的详细流程。方法中树中的每一个节点都保存了两个值：访问该节点的次数 N 和累计的奖励数量 Q，将两个数字相除就可以得到这个节点的平均奖励，也就是期望胜率。基于上面的分析，我们可以给出 UCT 的具体算法。

算法 UCT 算法

function UCTSEARCH(s_0)
  创建基于状态 s_0 的节点 v_0
  while 有时间和资源继续计算 do
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY(s(v_l))
    BACKUP(v_l, Δ)
  return a(BESTCHILD(v_0, 0))
-function TREEPOLICY(v)
  while v 不是树的叶子节点 do
    if v 没有被充分扩展 then
      return EXPAND(v)

```
    else
        v ←BESTCHILD(v, Cp)
    return v
-function EXPAND(v)
    从节点 v 未尝试的行动中选择一个行动 a
    为 v 增添一个叶子节点 v' 使得 s(v') = f(s(v), a)， a(v') = a
    return v'
-function BESTCHILD(v, c)
    return argmax_{v'∈children of v} Q(v')/N(v') + c√(2 ln N(v)/N(v'))
-function DEFAULTPOLICY(s)
    while s 是非终结状态 do
        根据策略选择行动 a
        得到下一个状态： s ← f(s, a)
    return reward for state s
-function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← v 的父节点
```

对于像围棋这样的零和博弈问题，对弈双方的得分互为相反数，因此在回报结果回传的过程中，同样需要对上一层的父节点的回报取反，如果结果是我方胜利，且给我方的记分是 1 分，那么给对方的记分就应该是 -1 分。

13.1.4 策略价值模型

前面我们介绍了 MCTS 的主要流程，下面就来介绍它是如何被应用到 AlphaZero 中的。从前面的介绍中读者已经了解了 UCT 的算法流程，对 AlphaZero 来说，它的主体流程与其类似，但是在细节上有以下两个不同。

（1）在选择步骤上，AlphaZero 使用了不同的公式对重要性进行判断：

$$U(s,a) = Q(s,a) + c_{\text{puct}} P(s,a) \frac{\sqrt{n}}{1+n_a}$$

其中 $Q(s,a)$ 为处于状态 s、行动 a 下的平均价值积累量，这个值由当前节点的所有叶

子节点的估计价值统计产生。$P(\boldsymbol{s},\boldsymbol{a})$ 是在状态 $\boldsymbol{s}$ 下执行行动 $\boldsymbol{a}$ 的概率，这个值由策略模型产生。

（2）在仿真这一步，AlphaZero 中并没有像 UCT 那样使用快速策略的生成招法，而是使用复杂的深层网络直接预测当前棋局的期望结果，这就像价值网络模型预测长期回报一样，如果游戏规则中获胜方获得 1 分，失败方获得 -1 分，那么网络预测的结果就应该在 -1 到 1 之间。从前面的学习经验来看，这种方法和仿真方法相比方差较小。蒙特卡罗法可以精确地得到一次采样的结果，但是这种方法同其他使用完整轨迹的方法一样，具有高方差的特点，不充分的采样会使价值期望产生波动；而价值网络使用了函数近似的方法，这种方法会存在一定的偏差，但同时会减少结果的波动。如果价值网络可以达到足够的精度，那么它的实际效果就会优于蒙特卡罗模拟。

当完成了对结果的估计后，这个结果会回传给所有的父节点。不同于前面提到的强化学习问题，这里我们不需要对价值进行打折，只要将估计值直接回传即可，过程如图 13-5 所示。

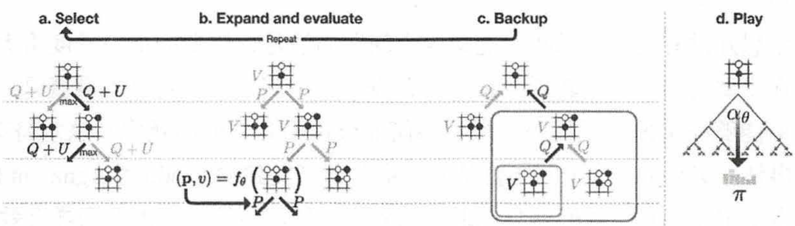


图 13-5　AlphaZero 的 MCTS 实现方式

我们发现，实现 AlphaZero 的 MCTS 需要两个模型：策略模型和价值模型，分别用 $\pi(\boldsymbol{a}|\boldsymbol{s};\theta)$ 和 $v(\boldsymbol{s};\boldsymbol{w})$ 表示。下面笔者就来分析两个模型的结构。

两个模型的输入相同，都是以棋盘为相对位置的特征。由于围棋的棋盘由 19×19 的交叉点组成，因此模型输入的长和宽都是 19。与 Atari 游戏类似，围棋也包含另外一个维度：时间。通过对比不同时刻棋盘的信息，我们可以知道棋盘最新落子的位置，这样我们可以更好地将落子位置这个信息考虑在内。因此，AlphaZero 考虑当前棋局前八招的招法信息，同时为了区分敌我双方，双方的棋子被拆分到不同的通道中，这样就有了 16 个通道的信息，加上一个用于判断是否为己方出招的通道，输入一共由 17 个通道组成。一个棋局的输入总大小为 $17\times19\times19$。

由于围棋的棋盘存在空间上的相邻关系和时间上的相邻关系，从两个维度来看它们都具有一定的局部相关性，因此我们可以使用卷积神经网络将这些特征提取出来。同时，由于两个模型都需要提取棋盘特征，因此这部分的运算可以共享，所以两个模型的前几层是共享的。策略模型的输出维度是 19×19，和棋盘的维度相同，每一个输出值

仅供非商业用途或交流学习使用

表示了棋子落在对应位置的概率。价值模型只输出一个值，用于表示当前局面下的结果估计。

从输出结果可以看出，策略模型相当于一个 361 类的分类，价值模型是一个回归模型，因此可以根据二者的模型类型定义对应的目标函数。策略模型的目标函数是一个 Cross Entropy Loss 函数，价值模型的目标函数是一个平方损失函数，这两个目标函数都是十分经典的函数。对于一个输入棋局来说，它的目标值是什么呢？

对于价值模型来说，目标值相对容易考虑，我们就以一局比赛结束后的结果作为整局比赛所有状态的目标输出。我们可以简单地认为如果一局对弈失败，那么整局所有招式都有问题，都应该给予一定的惩罚。这样的做法存在一定的不合理性，对棋类游戏有一定了解的读者一定会认为这种判定方法存在很大的问题。一局对弈持续的招式数可能很多，其中有的招式很优秀，有的招式很糟糕。博弈的结果往往和木桶原理相近，一局失利并不等于没有走出精妙的招式，只是存在一定的失误，那么决定棋局最终结果的应该是少数的几个招式，而不是所有的招式，我们应该只对个别招式进行惩罚，而不是惩罚所有招式。

这个思想并没有错，如果我们能将这个问题搞清楚，模型的学习进度将大幅提升。但如果站在模型的角度，或者说，将自己置身于某个问题中：自己本身就是一个新手，还不能保证能够完美地完成任务，那又如何能分析出其中的问题呢？这本身就是一个有挑战的事情。实际上，这个问题就是前面章节曾经提到的 Credit Assignment Problem。既然这件事情很难办到，我们就只能退而求其次，使用不那么完美但同样有效的方法。

以上是从悲观的角度分析最优方法的不可行，从乐观的方向分析，我们可以通过大数据将不同招式的结果期望区分开。假设经过足够多次数的对弈，同一个招式会在不同的棋局中出现，那么对那些不好的招式来说，在它们出现的棋局中，除它们之外，即使其他的招式都非常好，获胜的概率也相对较低；而对那些较好的招式来说，在它们出现的棋局中，除它们之外，其他招式只要不太差，获胜的概率会更高。这样经过更多局面的演绎，好招式和坏招式逐渐区分开，它们的结果期望也就有了差异，模型就能学到有用的信息。所以在大数据的环境下，这个问题可以得到一定程度的解决，读者也不用对此太担心。

对策略模型来说，目标值是 MCTS 的结果，也就是经过自我对弈分析后得到的结果。这个方法和 Q-Learning 有一些相似的地方，它们在计算目标值时都使用了模型自身计算的信息。从直觉上考虑，策略模型和价值模型接近最优时，经过自我对弈得到的结果应该比直接由策略模型得到的结果更严谨更可靠；而价值模型可以通过棋局结果优化提升，因此在学习的过程中，策略模型和价值模型总体上都在不断地提升，这样经过 MCTS 计算得到的行动会逐渐强于策略模型直接得到的行动。可以认为，以 MCTS 的结果作为目标值，能够提升模型策略，而再一次遇到类似的状态时，新的策略模型

仅供非商业用途或交流学习使用

经过 MCTS 的计算后可以获得更优秀的行动，通过这个机制，模型可以做到自我提升，从而摆脱训练数据的束缚。

在 AlphaZero 的前一代实现 AlphaGo 中，策略模型和价值模型仍然需要使用训练数据进行训练。实际上这是一种很不错的方法，笔者在第 8 章中也介绍了将准备好的数据放入 Replay Buffer 进行训练的方法，这种方法取得了不错的效果。使用训练数据进行训练虽然可以快速提高模型的能力，但是它也存在一定的隐患。因为围棋游戏毕竟是一个尚未完美解决的问题，真正的最优策略仍未找寻到，所以我们并不知道某一棋局的最优招法。虽然我们可以找到一些优秀的对弈棋谱，但是谁也无法证明这些棋谱的每一招都是最优的，于是对于采用监督学习的模型来说，棋谱中的优秀招式和不好的招式统统被学习到了模型中，模型的水平很难超越人类的既有水平。若使用 MCTS 提升模型的方法可以成立，那么虽然在一开始模型的表现会很差，但随着不断地学习，模型会变得越来越好，这样也就有了超越人类的可能。

模型完整的目标函数可以表示为

$$L = \sum_t (v(\boldsymbol{s}_t; \boldsymbol{w}) - \boldsymbol{z}_t)^2 - \tilde{\pi}_t \log \pi(\boldsymbol{a}|\boldsymbol{s}_t; \theta) + c\|\theta\|^2 + c\|\boldsymbol{w}\|^2$$

目标中的 t 表示训练过程中的样本编号，$\boldsymbol{z}_t$ 表示棋局对弈的最终结果，玩家胜利为 1 分，失败为 -1 分；$\tilde{\pi}$ 表示经过 MCTS 计算得到的行动概率。公式最后两项是对参数的正则约束，c 为控制正则项的权重。可以看出，我们需要在对弈过程中使用 Replay Buffer 保存棋局状态、MCTS 行动结果对应的概率，以及棋局的最终结果三个内容。完整的对弈过程如图 13-6 所示。

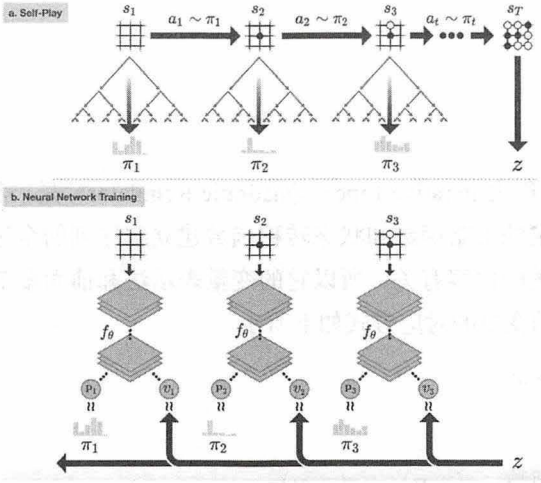


图 13-6　自我对弈训练过程示意图

仅供非商业用途或交流学习使用

前面关于模型提升的思想也只是一些直观的想法，并没有严格的理论证明，所以随着学习不断推进，模型效果下降的情况是可能发生的。为了解决这个问题，AlphaZero 中还包含了第三个模块，这个模块的存在使模型的单调提升变得更有保证。

13.1.5 模型的对决

为了让模型能够拥有接近单调提升的性质，AlphaZero 中包含了模型之间对决的环节。俗话说“是骡子是马，拉出来遛遛”，对于如此复杂的问题，最好的检验方法就是在真实对弈中看效果。于是每完成一轮迭代，新的模型都要和旧的模型进行一定轮棋局的对弈。新的模型只有获得更多的胜利（超过 55% 的胜率）才被认为超过了旧模型，才能替换旧的模型，否则就抛弃这个新模型，重新进行新的训练。

虽然这个思想并不复杂，但其中包含了大量的计算，而且算法中的模型比较大，在对决时需要将其部署到多台机器上同时进行多局对弈，因此其实现需要工程上的考量。相对而言，这个方法在思想上比较简单，而 AlphaZero 中的自我博弈已经具备一定的使策略提升的功能，因此这里的设计可以简单一些。

以上就是 AlphaZero 的核心流程。它的三个核心模块：自我对弈、深层模型和新旧模型对决组成了完整的模型，为棋类游戏构建了一个新的框架范本。很多与围棋形式接近的棋类也可以应用这个框架实现对应的策略。在论文中作者也介绍了 AlphaZero 在国际象棋和日本将棋上的应用，很多人也尝试在其他棋类上应用，并且收到了很好的效果，这也证明了通用学习框架对人工智能发展起到的作用。更多关于模型训练的细节请读者参阅相关论文，这里不再赘述。

13.2 iLQR

iLQR 算法的全称为 iterative Linear Quadratic Regulator，是一种 Model-based 的规划方法，它使用确定的策略函数和状态转移函数建立起序列的全过程。由于这部分知识最早与控制论相关的内容有关，所以它的变量表示法和前面章节中强化学习的表示方法有所不同，它的变量的对应方式如下所示。

观测状态：$\boldsymbol{s} \to \boldsymbol{x}$。

行动：$\boldsymbol{a} \to \boldsymbol{u}$。

回报（损失）函数：$r(\boldsymbol{s}, \boldsymbol{a}) \to c(\boldsymbol{x}, \boldsymbol{u})$。

状态转移函数：$\boldsymbol{s}_{t+1} \sim p(\boldsymbol{s}_{t+1} | \boldsymbol{s}_t, \boldsymbol{a}_t) \to \boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t)$。

仅供非商业用途或交流学习使用

在控制论中，损失函数 $c(\boldsymbol{x}, \boldsymbol{u})$ 表示执行行动后产生的代价。因此强化学习的目标——“最大化长期回报”变成了“最小化长期损失”。它要解决的问题形式如下所示：

$$\min_{\boldsymbol{u}_1,\ldots,\boldsymbol{u}_T} c(\boldsymbol{x}_1, \boldsymbol{u}_1) + c(\boldsymbol{x}_2, \boldsymbol{u}_2) + \cdots + c(\boldsymbol{x}_T, \boldsymbol{u}_T)$$

因为状态转移函数已知，所以目标函数也可以写作

$$\min_{\boldsymbol{u}_1,\ldots,\boldsymbol{u}_T} c(\boldsymbol{x}_1, \boldsymbol{u}_1) + c(f(\boldsymbol{x}_1, \boldsymbol{u}_1), \boldsymbol{u}_2) + \cdots + c(f(f(\ldots)\ldots), \boldsymbol{u}_T)$$

第 6 章介绍了状态转移概率已知的问题。那么这个问题和第 6 章介绍的问题有什么区别呢？第 6 章介绍的问题在状态和行动空间上都是离散的，而本章介绍的问题在状态和行动空间上都是连续的，而且状态转移模型是一个确定的模型。在现实中我们也会经常遇到这样的问题，例如操纵一个机器人在一个环境中实现某个任务，由于任务的全部物理过程是已知的，所以状态转移函数是确定的；同时，我们也可以构建确定的策略函数，这样完整的交互序列可以由公式表示。可以看出这个问题和第 6 章提到的蛇棋问题的形式不同，当然二者的算法也不一样。

13.2.1 线性模型的求解法

我们先从一个比较简单的问题入手，介绍在计算过程上相对简单一些的算法 LQR。假设损失函数是一个二次函数，而状态转移函数是一个一次函数，那么两个公式可以写作

$$c(\boldsymbol{x}_t, \boldsymbol{u}_t) = \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix}^{\mathrm{T}} \boldsymbol{C} \begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix} + \begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix}^{\mathrm{T}} \boldsymbol{c}$$

$$f(\boldsymbol{x}_t, \boldsymbol{u}_t) = F\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix} + f$$

这样定义后，目标函数中唯一未知的变量就是行动 $\boldsymbol{u}_t$，我们可以通过求解公式的极值找到最优的行动。为了让问题看上去不那么复杂，我们先不考虑较长序列的问题，而是把行动轨迹限定在两个时间段内，那么整个序列为 $(\boldsymbol{x}_1, \boldsymbol{u}_1, \boldsymbol{x}_2, \boldsymbol{u}_2)$，我们要推导出当 $\boldsymbol{x}_1$ 已知时使损失函数最小化的行动，即求出公式的最优解

$$\min_{\boldsymbol{u}_1,\boldsymbol{u}_2} Q = c(\boldsymbol{x}_1, \boldsymbol{u}_1) + c(f(\boldsymbol{x}_1, \boldsymbol{u}_1), \boldsymbol{u}_2)$$

仅供非商业用途或交流学习使用

如果可以从这个问题中找到规律，就可以把求解公式推广到更长的行动序列上。将公式完全展开，可以得到

$$
\begin{aligned}
Q &= \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}}\boldsymbol{C}\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{u}_1\end{bmatrix}+\begin{bmatrix}\boldsymbol{x}_t\\ \boldsymbol{u}_t\end{bmatrix}^{\mathrm{T}}\boldsymbol{c}+\frac{1}{2}\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{u}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{C}\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{u}_2\end{bmatrix}+\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{u}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{c}\\
&= \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}}\boldsymbol{C}\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{u}_1\end{bmatrix}+\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}}\boldsymbol{c}+\frac{1}{2}\begin{bmatrix}f(\boldsymbol{x}_1,\boldsymbol{u}_1)\\ \boldsymbol{u}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{C}\begin{bmatrix}f(\boldsymbol{x}_1,\boldsymbol{u}_1)\\ \boldsymbol{u}_2\end{bmatrix}+\begin{bmatrix}f(\boldsymbol{x}_1,\boldsymbol{u}_1)\\ \boldsymbol{u}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{c}
\end{aligned}
$$

公式中的 $\boldsymbol{C}$ 和 $\boldsymbol{c}$ 可以展开为

$$
\boldsymbol{C}=\begin{bmatrix}\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{x}_T},\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\\ \boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{x}_T},\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\end{bmatrix}
$$

$$
\boldsymbol{c}=\begin{bmatrix}\boldsymbol{c}_{\boldsymbol{x}_T}\\ \boldsymbol{c}_{\boldsymbol{u}_T}\end{bmatrix}
$$

其中矩阵 $\boldsymbol{C}$ 是一个对称矩阵，也就是说 $\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}=\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{x}_T}$。$\boldsymbol{F}$ 也可以展开为

$$
\boldsymbol{F}=[\boldsymbol{F}_{\boldsymbol{x}_T},\boldsymbol{F}_{\boldsymbol{u}_T}]
$$

公式待求的是 $\boldsymbol{u}_1,\boldsymbol{u}_2$ 的值，对其分别进行求导，并使导数为 0，就可以得到函数取得极值时两个变量的解，首先对 $\boldsymbol{u}_2$ 求导可以得到

$$
\nabla_{\boldsymbol{u}_2}\boldsymbol{Q}=\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}f(\boldsymbol{x}_1,\boldsymbol{u}_1)+\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\boldsymbol{u}_2+\boldsymbol{c}_{\boldsymbol{u}_T}^{\mathrm{T}}
$$

再对 $\boldsymbol{u}_1$ 进行求导，可以得到

$$
\nabla_{\boldsymbol{u}_1}\boldsymbol{Q}=\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{x}_1+\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\boldsymbol{u}_1+\boldsymbol{c}_{\boldsymbol{u}_T}^{\mathrm{T}}+\boldsymbol{F}_{\boldsymbol{u}_T}^{\mathrm{T}}(\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{x}_T}f(\boldsymbol{x}_1,\boldsymbol{u}_1)+\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{u}_2+\boldsymbol{c}_{\boldsymbol{x}_T}^{\mathrm{T}})
$$

虽然我们得到了这两个变量的导数，但是它们的形式十分烦琐。两个行动的求解计算已经如此复杂，两个以上行动的计算应该会更复杂，而且每一时刻行动的导数计算公式都不相同，因此我们不能使用这样的方法对公式直接展开求解，而是需要使用一种更简洁且易操作的方法实现。这就是 LQR 方法要解决的问题。

LQR 方法包含反向计算部分和前向计算部分，其中反向计算用于确定计算每一个时刻行动的参数值，而前向计算则根据初始状态和反向计算的参数得到所有时刻的行动。反向计算比较复杂，它通过迭代的形式完成计算，每一个迭代中，我们要完成一个

仅供非商业用途或交流学习使用

时刻行动参数的计算，当这些参数计算完成后，我们就可以通过前向计算得到具体的行动值。迭代内的推导过程分为如下几个步骤。

（1）根据 T 时刻的目标公式对 $\boldsymbol{u}$ 求导，计算使函数最优的 $\boldsymbol{u}$ 值。

（2）用 $\boldsymbol{x}$ 替换 $\boldsymbol{u}$，得到关于 $\boldsymbol{x}$ 的优化公式。

（3）将公式转换成 $t-1$ 时刻变量 $\boldsymbol{x}$、$\boldsymbol{u}$ 的形式。

下面就来介绍具体过程。第一步，在 $T=2$ 时刻通过最小化 $C(\boldsymbol{x}_2,\boldsymbol{u}_2)$ 求出 $\boldsymbol{u}_2$，根据公式，可以得到

$$Q(\boldsymbol{x}_2,\boldsymbol{u}_2)=\frac{1}{2}\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{u}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{C}\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{u}_2\end{bmatrix}+\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{u}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{c}$$

对其进行求导，可以得到

$$\nabla_{\boldsymbol{u}_2}Q(\boldsymbol{x}_2,\boldsymbol{u}_2)=\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{x}_2+\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\boldsymbol{u}_2+\boldsymbol{c}_{\boldsymbol{u}_T}^{\mathrm{T}}$$

令函数的导数为 0，可以得到关于 $\boldsymbol{u}_2$ 的公式：

$$\boldsymbol{u}_2=-\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}^{-1}(\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}^{\mathrm{T}}\boldsymbol{x}_2+\boldsymbol{c}_{\boldsymbol{u}_T}^{\mathrm{T}})$$

为了简化表达式，用变量 $\boldsymbol{K}_2$ 和 $\boldsymbol{k}_2$ 代替公式中的表达式，可以得到

$$\boldsymbol{K}_2=-\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}^{-1}\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}^{\mathrm{T}}$$
$$\boldsymbol{k}_2=-\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}^{-1}\boldsymbol{c}_{\boldsymbol{u}_T}^{\mathrm{T}}$$

于是 $\boldsymbol{u}_2$ 的值可以表示为关于 $\boldsymbol{x}_2$ 的公式：

$$\boldsymbol{u}_2=\boldsymbol{K}_2\boldsymbol{x}_2+\boldsymbol{k}_2$$

这样第一步就完成了。第二步的目标是得到关于 $\boldsymbol{x}_2$ 的优化公式，将上面关于 $\boldsymbol{u}_2$ 的公式代入损失函数中，可以得到

$$V(\boldsymbol{x}_2)=\min Q(\boldsymbol{x}_2,\boldsymbol{u}_2)=\frac{1}{2}\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{K}_2\boldsymbol{x}_2+\boldsymbol{k}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{C}\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{K}_2\boldsymbol{x}_2+\boldsymbol{k}_2\end{bmatrix}+\begin{bmatrix}\boldsymbol{x}_2\\ \boldsymbol{K}_2\boldsymbol{x}_2+\boldsymbol{k}_2\end{bmatrix}^{\mathrm{T}}\boldsymbol{c}$$

仅供非商业用途或交流学习使用

将其展开可以得到

$$
\begin{aligned}
&= \frac{1}{2}\boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{x}_T}\boldsymbol{x}_2 + \frac{1}{2}\boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{K}_2\boldsymbol{x}_2 + \frac{1}{2}\boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{x}_T}\boldsymbol{x}_2 + \frac{1}{2}\boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\boldsymbol{K}_2\boldsymbol{x}_2 \\
&+ \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{K}_T^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_t,\boldsymbol{u}_T}\boldsymbol{k}_T + \frac{1}{2}\boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{X}_T,\boldsymbol{U}_T}\boldsymbol{k}_T + \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{c}_{\boldsymbol{x}_T} + \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{c}_{\boldsymbol{u}_T} + \text{const}
\end{aligned}
$$

这个公式展开后也非常复杂，我们同样需要对公式中的变量进行替换，使用变量 $\boldsymbol{V}_2$ 和 $\boldsymbol{v}_2$，可以得到

$$
\begin{aligned}
\boldsymbol{V}_2 &= \boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{x}_T} + \boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{K}_2 + \boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{x}_T} + \boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\boldsymbol{K}_2 \\
\boldsymbol{v}_2 &= \boldsymbol{c}_{\boldsymbol{x}_T} + \boldsymbol{C}_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{k}_2 + \boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_T} + \boldsymbol{K}_2^{\mathrm{T}}\boldsymbol{C}_{\boldsymbol{u}_T,\boldsymbol{u}_T}\boldsymbol{k}_2
\end{aligned}
$$

公式又重新变成了十分简洁的形式：

$$
V(\boldsymbol{x}_2) = \text{const} + \frac{1}{2}\boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{V}_2\boldsymbol{x}_2 + \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{v}_2
$$

这样第二步就完成了。接下来是第三步，即将目标函数转换成以 $\boldsymbol{x}_1, \boldsymbol{u}_1$ 表示的形式，将原公式中第二时刻的损失函数替换成前面计算的结果，就可以得到下面的形式：

$$
Q(\boldsymbol{x}_1, \boldsymbol{u}_1) = \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{C} \begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix} + \begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{c} + V(f(\boldsymbol{x}_1, \boldsymbol{u}_1))
$$

可以看出公式等号右边包含三个项目，第一项和第二项是 $t = 1$ 时刻的行动损失，第三项是前面计算得到的 $V(\boldsymbol{x}_2)$，将上面公式的第二项完全展开，可以得到

$$
\begin{aligned}
V(f(\boldsymbol{x}_1, \boldsymbol{u}_1)) = V(\boldsymbol{F}&\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix} + \boldsymbol{f}) \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{F}^{\mathrm{T}}\boldsymbol{V}_2\boldsymbol{F}\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix} \\
&+ \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{F}^{\mathrm{T}}\boldsymbol{V}_2\boldsymbol{f} + \frac{1}{2}\boldsymbol{f}\boldsymbol{V}_2\boldsymbol{F}\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix} + \begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{F}^{\mathrm{T}}\boldsymbol{v}_2 + \text{const}
\end{aligned}
$$

展开后的第二项和原公式中的前两项存在可合并的同类项，这样就可以做进一步的变量替换和整理，得到新的公式

$$
\begin{aligned}
\boldsymbol{Q}_1 &= \boldsymbol{C} + \boldsymbol{F}^{\mathrm{T}}\boldsymbol{V}_2\boldsymbol{F} \\
\boldsymbol{q}_1 &= \boldsymbol{c} + \boldsymbol{F}^{\mathrm{T}}\boldsymbol{V}_2\boldsymbol{f} + \boldsymbol{F}^{\mathrm{T}}\boldsymbol{v}_2
\end{aligned}
$$

仅供非商业用途或交流学习使用

$$Q(\boldsymbol{x}_1, \boldsymbol{u}_1) = \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{Q}_1 \begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix} + \begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{u}_1\end{bmatrix}^{\mathrm{T}} \boldsymbol{q}_1$$

实际上，到这里我们已经完成了一轮迭代的推演。我们发现上面的公式变成了只包含 $\boldsymbol{x}_1$ 和 $\boldsymbol{u}_1$ 的公式，可以使用第一步的计算方法求出使公式最优的 $\boldsymbol{u}_1$ 值：

$$\begin{aligned}
\nabla_{\boldsymbol{u}_1} Q(\boldsymbol{x}_1, \boldsymbol{u}_1) &= \boldsymbol{Q}_{\boldsymbol{x}_T, \boldsymbol{u}_T}^{\mathrm{T}} \boldsymbol{x}_1 + \boldsymbol{Q}_{\boldsymbol{u}_T, \boldsymbol{u}_T}^{\mathrm{T}} \boldsymbol{u}_1 + \boldsymbol{q}_{\boldsymbol{u}_T}^{\mathrm{T}} = 0 \\
\boldsymbol{u}_1 &= \boldsymbol{K}_1 \boldsymbol{x}_1 + \boldsymbol{k}_1 \\
\boldsymbol{K}_1 &= -\boldsymbol{Q}_{\boldsymbol{u}_T, \boldsymbol{u}_T}^{-1} \boldsymbol{Q}_{\boldsymbol{u}_T, \boldsymbol{x}_T} \\
\boldsymbol{k}_1 &= -\boldsymbol{Q}_{\boldsymbol{u}_T, \boldsymbol{u}_T}^{-1} \boldsymbol{q}_{\boldsymbol{u}_T}
\end{aligned}$$

而此时 $\boldsymbol{x}_1$ 已知，我们就可以直接得到 $\boldsymbol{u}_1$ 的值，然后根据状态转移函数得到 $\boldsymbol{x}_2$，再利用参数 K_2 和 k_2 进行计算得到行动 $\boldsymbol{u}_2$。这其实就是前向计算的过程。理解了长度为 2 的序列，再去推导长度为 N 的序列也变得不那么复杂，完整的算法过程如下。

算法 LQR 反向计算算法

$\boldsymbol{V}_{T+1} = 0$

$\boldsymbol{v}_{t+1} = 0$

For $t = T$ to 1;

$\boldsymbol{Q}_t = \boldsymbol{C} + \boldsymbol{F}^{\mathrm{T}} \boldsymbol{V}_{t+1} \boldsymbol{F}$

$\boldsymbol{q}_t = \boldsymbol{c} + \boldsymbol{F}^{\mathrm{T}} \boldsymbol{V}_{t+1} \boldsymbol{f} + \boldsymbol{F}^{\mathrm{T}} \boldsymbol{v}_{t+1}$

$$Q(\boldsymbol{x}_t, \boldsymbol{u}_t) = \frac{1}{2}\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix}^{\mathrm{T}} \boldsymbol{Q}_t \begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix} + \begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{u}_t\end{bmatrix}^{\mathrm{T}} \boldsymbol{q}_t$$

$\boldsymbol{K}_t = -\boldsymbol{Q}_{\boldsymbol{tu}_T, \boldsymbol{u}_T}^{-1} \boldsymbol{Q}_{\boldsymbol{tu}_T, \boldsymbol{x}_T}$

$\boldsymbol{k}_1 = -\boldsymbol{Q}_{\boldsymbol{tu}_T, \boldsymbol{u}_T}^{-1} \boldsymbol{q}_{\boldsymbol{tu}_T}$

$\boldsymbol{u}_t = \mathrm{argmin}_{\boldsymbol{u}_t} Q(\boldsymbol{x}_t, \boldsymbol{u}_t) = \boldsymbol{K}_t \boldsymbol{x}_t + \boldsymbol{k}_t$

$\boldsymbol{V}_t = \boldsymbol{Q}_{\boldsymbol{tx}_T, \boldsymbol{x}_T} + \boldsymbol{Q}_{\boldsymbol{tx}_T, \boldsymbol{u}_T} \boldsymbol{K}_t + \boldsymbol{K}_t^{\mathrm{T}} \boldsymbol{Q}_{\boldsymbol{tu}_T, \boldsymbol{x}_T} + \boldsymbol{K}_t^{\mathrm{T}} \boldsymbol{Q}_{\boldsymbol{tu}_T, \boldsymbol{u}_T} \boldsymbol{K}_t$

$\boldsymbol{v}_t = \boldsymbol{q}_{\boldsymbol{tx}_T} + \boldsymbol{Q}_{\boldsymbol{tx}_T, \boldsymbol{u}_T} \boldsymbol{k}_t + \boldsymbol{K}_t^{\mathrm{T}} \boldsymbol{Q}_{\boldsymbol{tu}_T} + \boldsymbol{K}_t^{\mathrm{T}} \boldsymbol{Q}_{\boldsymbol{tu}_T, \boldsymbol{u}_T} \boldsymbol{k}_t$

$$V(\boldsymbol{x}_t) = \mathrm{const} + \frac{1}{2} \boldsymbol{x}_t^{\mathrm{T}} \boldsymbol{V}_t \boldsymbol{x}_t + \boldsymbol{x}_t^{\mathrm{T}} \boldsymbol{v}_t$$

算法 LQR 前向计算算法

for $t = 1$ to T:

$\boldsymbol{u}_t = \boldsymbol{K}_t \boldsymbol{x}_t + \boldsymbol{k}_t$

$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t)$

13.2.2 非线性模型的解法

本节我们介绍 iLQR 算法。13.2.1 节介绍的方法存在一个假设：损失函数为二次项函数，状态转移函数为一次函数。然而在很多真实场景下，我们得到的损失函数和状态转移函数并不一定满足上面的条件，例如它们并不是二次函数或者一次函数，而且问题的形式往往也会有一些不同的设定，例如最后一个状态并不需要给出行动，而是采用另一个损失函数直接计算最终状态的花费。由于 LQR 的推导相对复杂，在此我们可以借助前面的经验，对问题进行重新求解。

我们会对最终状态单独设定一个花费函数 $c_N(\boldsymbol{x})$，用于计算其花费，而其他时刻的花费仍然使用 $c(\boldsymbol{x}, \boldsymbol{u})$ 计算，这样 N 个时刻总体的花费就可以计算成

$$J(\boldsymbol{x}_0, \boldsymbol{U}) = \sum_{t=0}^{N-1} c(\boldsymbol{x}_t, \boldsymbol{u}_t) + c_N(\boldsymbol{x}_N)$$

其中 $\boldsymbol{U}$ 代表序列中执行的全体行动 $\{\boldsymbol{u}_t\}_{t=1}^N$。为了使用 13.2.1 节的方法进行求解，我们需要将问题转化为类似的形式，这就要用到泰勒展开这个工具。对于任意的损失函数和状态转移函数，我们可以将其近似地展开，首先是损失函数：

$$\begin{aligned}
c(\boldsymbol{x}_t, \boldsymbol{u}_t) &= c(\hat{\boldsymbol{x}}_t + \boldsymbol{\delta x}_t, \hat{\boldsymbol{u}}_t + \boldsymbol{\delta u}_t) \\
&\simeq c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) + \nabla_{\boldsymbol{x}_t, \boldsymbol{u}_t} c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}^{\mathrm{T}} \nabla^2_{\boldsymbol{x}_t, \boldsymbol{u}_t} c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix} \\
&=\simeq c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) + \boldsymbol{c} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}^{\mathrm{T}} \boldsymbol{C} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}
\end{aligned}$$

假设 $\boldsymbol{x}_t, \boldsymbol{u}_t$ 和 $\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t$ 足够接近，那么两者的梯度也可以近似，我们因此可以得到

$$\begin{aligned}
\nabla_{\boldsymbol{x}_t, \boldsymbol{u}_t} c(\boldsymbol{x}_t, \boldsymbol{u}_t) &\simeq \nabla_{\boldsymbol{x}_t, \boldsymbol{u}_t} c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \\
\nabla^2_{\boldsymbol{x}_t, \boldsymbol{u}_t} c(\boldsymbol{x}_t, \boldsymbol{u}_t) &\simeq \nabla^2_{\boldsymbol{x}_t, \boldsymbol{u}_t} c(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)
\end{aligned}$$

同理，我们可以对状态转移函数进行展开：

$$\begin{aligned}
f(\boldsymbol{x}_t, \boldsymbol{u}_t) &= f(\hat{\boldsymbol{x}}_t + \boldsymbol{\delta x}_t, \hat{\boldsymbol{u}}_t + \boldsymbol{\delta u}_t) \\
&\simeq f(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) + \nabla_{\boldsymbol{x}_t, \boldsymbol{u}_t} f(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}^{\mathrm{T}} \nabla^2_{\boldsymbol{x}_t, \boldsymbol{u}_t} f(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}
\end{aligned}$$

$$= f(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t) + \boldsymbol{F} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}^{\mathrm{T}} \boldsymbol{F} \begin{bmatrix} \boldsymbol{\delta x}_t \\ \boldsymbol{\delta u}_t \end{bmatrix}$$

同损失函数类似，我们可以得到近似梯度的公式：

$$\nabla_{\boldsymbol{\delta x}_t, \boldsymbol{\delta u}_t} f(\boldsymbol{x}_t, \boldsymbol{u}_t) \simeq \nabla_{\boldsymbol{x}_t, \boldsymbol{u}_t} f(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$$
$$\nabla^2_{\boldsymbol{\delta x}_t, \boldsymbol{\delta u}_t} f(\boldsymbol{x}_t, \boldsymbol{u}_t) \simeq \nabla^2_{\boldsymbol{x}_t, \boldsymbol{u}_t} f(\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t)$$

经过这样的转换，我们发现当函数的表示形式发生变化时，我们从直接求解 $\boldsymbol{x}_t, \boldsymbol{u}_t$，变成了已知序列 $\{\hat{\boldsymbol{x}}_t, \hat{\boldsymbol{u}}_t\}_{t=1}^N$ 的情况下求解 $\boldsymbol{\delta x}$ 和 $\boldsymbol{\delta u}$。转换后的求解并不困难，我们可以初始化一个行动序列 $\{\hat{\boldsymbol{u}}_t\}_{t=1}^N$，然后从初始状态 $\boldsymbol{s}_1$ 出发通过状态转移函数得到这样的序列。为了展示公式的推导过程，我们还是假设一个相对简单的序列 $\{\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{u}}_2, \hat{\boldsymbol{x}}_3\}$，从这个序列上得到的公式很容易应用到解决复杂的问题上。

下面就来使用和 13.3.1 节类似的求解方法进行计算。首先对 $\boldsymbol{x}_3$ 进行优化，可以直接得到

$$V(\boldsymbol{x}_3) = c_N(\boldsymbol{x}_3)$$
$$\nabla_{\boldsymbol{x}} V(\boldsymbol{x}_3) = \nabla_{\boldsymbol{x}} c_N(\boldsymbol{x}_3)$$
$$\nabla_{\boldsymbol{xx}} V(\boldsymbol{x}_3) = \nabla_{\boldsymbol{xx}} c_N(\boldsymbol{x}_3)$$

由于我们直接得到了关于 $\boldsymbol{x}_3$ 的目标函数，所以直接进行 13.2.1 节中反向计算的第 3 步，将目标函数表示为 $\boldsymbol{x}_2, \boldsymbol{u}_2$ 的形式，此时的目标函数为

$$Q(\boldsymbol{x}_2, \boldsymbol{u}_2) = c(\hat{\boldsymbol{x}}_2 + \boldsymbol{\delta x}_2, \hat{\boldsymbol{u}}_2 + \boldsymbol{\delta u}_2) + V(f(\hat{\boldsymbol{x}}_2 + \boldsymbol{\delta x}_2, \hat{\boldsymbol{u}}_2 + \boldsymbol{\delta u}_2))$$

对其求导，将公式右边的第一项简写为 $\boldsymbol{c}$，第二项简写为 $\boldsymbol{f}$，可以得到上面函数的导数：

$$\nabla_{\boldsymbol{\delta x}_2} \boldsymbol{Q} = \nabla_{\hat{\boldsymbol{x}}_2} \boldsymbol{c} + \nabla_{\hat{\boldsymbol{x}}_3} \boldsymbol{V} \nabla_{\boldsymbol{\delta x}_2} \boldsymbol{f}$$
$$\nabla_{\boldsymbol{\delta u}_2} \boldsymbol{Q} = \nabla_{\hat{\boldsymbol{u}}_2} \boldsymbol{c} + \nabla_{\hat{\boldsymbol{x}}_3} \boldsymbol{V} \nabla_{\boldsymbol{\delta u}_2} \boldsymbol{f}$$

继续求二阶导，可以得到

$$\nabla_{\boldsymbol{\delta x}_2 \boldsymbol{\delta x}_2} \boldsymbol{Q} = \nabla_{\boldsymbol{\delta x}_2} (\nabla_{\hat{\boldsymbol{x}}_2} \boldsymbol{c} + \nabla_{\hat{\boldsymbol{x}}_3} \boldsymbol{V} \nabla_{\boldsymbol{\delta x}_2} \boldsymbol{f})$$

$$
\begin{aligned}
&= \nabla_{\hat{\boldsymbol{x}}_2\hat{\boldsymbol{x}}_2}\boldsymbol{c} + \nabla_{\delta\boldsymbol{x}_2}\boldsymbol{f}^{\mathrm{T}}\nabla_{\hat{\boldsymbol{x}}_3\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{x}_2}\boldsymbol{f} + \nabla_{\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{x}_2\delta\boldsymbol{x}_2}\boldsymbol{f} \\
\nabla_{\delta\boldsymbol{u}_2\delta\boldsymbol{x}_2}\boldsymbol{Q} &= \nabla_{\delta\boldsymbol{u}_2}(\nabla_{\hat{\boldsymbol{x}}_2}\boldsymbol{c} + \nabla_{\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{x}_2}\boldsymbol{f}) \\
&= \nabla_{\hat{\boldsymbol{u}}_2\hat{\boldsymbol{x}}_2}\boldsymbol{c} + \nabla_{\delta\boldsymbol{u}}\boldsymbol{f}^{\mathrm{T}}\nabla_{\hat{\boldsymbol{x}}_3\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{x}}\boldsymbol{f} + \nabla_{\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{u}_2\delta\boldsymbol{x}_2}\boldsymbol{f} \\
\nabla_{\delta\boldsymbol{u}_2\delta\boldsymbol{u}_2}\boldsymbol{Q} &= \nabla_{\delta\boldsymbol{u}_2}(\nabla_{\hat{\boldsymbol{u}}_2}\boldsymbol{c} + \nabla_{\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{u}_2}\boldsymbol{f}) \\
&= \nabla_{\hat{\boldsymbol{u}}_2\hat{\boldsymbol{u}}_2}\boldsymbol{c} + \nabla_{\delta\boldsymbol{u}_2}\boldsymbol{f}^{\mathrm{T}}\nabla_{\hat{\boldsymbol{x}}_3\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{u}_2}\boldsymbol{f} + \nabla_{\hat{\boldsymbol{x}}_3}\boldsymbol{V}\nabla_{\delta\boldsymbol{u}_2\delta\boldsymbol{u}_2}\boldsymbol{f}
\end{aligned}
$$

知道了这些导数，就可以进行 13.2.1 节中反向计算的第 1 步，求出当前时刻的最优值 $\boldsymbol{u}$，这里我们实际求出的是最优的 $\delta\boldsymbol{u}$，令 $\nabla_{\delta\boldsymbol{u}_2}\boldsymbol{Q}=0$，可以得到

$$
\delta\boldsymbol{u}_2 = -\nabla_{\boldsymbol{x}_T,\boldsymbol{u}_T}Q^{-1}(\nabla_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{Q}^{\mathrm{T}}\delta\boldsymbol{x}_2 + \nabla_{\boldsymbol{u}_T}\boldsymbol{Q}^{\mathrm{T}})
$$

将公式进行整理，可以得到

$$
\begin{aligned}
\boldsymbol{K}_2 &= -\nabla_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{Q}^{-1}\nabla_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{Q}^{\mathrm{T}} \\
\boldsymbol{k}_2 &= -\nabla_{\boldsymbol{x}_T,\boldsymbol{u}_T}\boldsymbol{Q}^{-1}\nabla_{\boldsymbol{u}_T}\boldsymbol{Q}^{\mathrm{T}} \\
\boldsymbol{u}_2 &= \hat{\boldsymbol{u}}_2 + \delta\boldsymbol{u}_2 = \hat{\boldsymbol{u}}_2 + \boldsymbol{K}_2\delta\boldsymbol{x}_2 + \boldsymbol{k}_2
\end{aligned}
$$

接下来就是 13.2.1 节反向计算的第 2 步，整理公式得到关于 $\boldsymbol{x}_2$ 的目标函数：

$$
\begin{aligned}
V(\boldsymbol{x}_2) &= Q(\hat{\boldsymbol{x}}_2 + \delta\boldsymbol{x}_2, \hat{\boldsymbol{u}}_2 + \boldsymbol{K}_2\delta\boldsymbol{x}_2 + \boldsymbol{k}_2) \\
&= c(\hat{\boldsymbol{x}}_2 + \delta\boldsymbol{x}_2, \hat{\boldsymbol{u}}_2 + \boldsymbol{K}_2\delta\boldsymbol{x}_2 + \boldsymbol{k}_2) + V(f(\hat{\boldsymbol{x}}_2 + \delta\boldsymbol{x}_2, \hat{\boldsymbol{u}}_2 + \boldsymbol{K}_2\delta\boldsymbol{x}_2 + \boldsymbol{k}_2))
\end{aligned}
$$

同样，我们可以求出公式的一阶导和二阶导供后续使用：

$$
\begin{aligned}
\nabla_{\boldsymbol{x}}\boldsymbol{V} &= \nabla_{\boldsymbol{x}}\boldsymbol{c} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}}\boldsymbol{c} + \nabla_{\boldsymbol{x}}\boldsymbol{V} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}}\boldsymbol{V} \\
&= \nabla_{\boldsymbol{x}}\boldsymbol{c} + \nabla_{\boldsymbol{x}}\boldsymbol{V} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}}\boldsymbol{c} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}}\boldsymbol{V} \\
&= \nabla_{\boldsymbol{x}}\boldsymbol{Q} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}}\boldsymbol{Q} \\
\nabla_{\boldsymbol{x}\boldsymbol{x}}\boldsymbol{V} &= \nabla_{\boldsymbol{x}}(\nabla_{\boldsymbol{x}}\boldsymbol{Q} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}}\boldsymbol{Q}) \\
&= \nabla_{\boldsymbol{x}\boldsymbol{x}}\boldsymbol{Q} + \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}\boldsymbol{x}}\boldsymbol{Q}
\end{aligned}
$$

有些文献也喜欢把这两个公式进行一定的变换，将其变为

$$
\begin{aligned}
\nabla_{\boldsymbol{x}}\boldsymbol{V} &= \nabla_{\boldsymbol{x}}\boldsymbol{Q} - \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}\boldsymbol{u}}\boldsymbol{Q}\boldsymbol{k}_2 \\
\nabla_{\boldsymbol{x}\boldsymbol{x}}\boldsymbol{V} &= \nabla_{\boldsymbol{x}\boldsymbol{x}}\boldsymbol{Q} - \boldsymbol{K}_2^{\mathrm{T}}\nabla_{\boldsymbol{u}\boldsymbol{u}}\boldsymbol{Q}\boldsymbol{K}_2
\end{aligned}
$$

感兴趣的读者可以证明前后两组公式的等价性。这样，一个完整循环的计算内容就结束了。对于更长的循环，我们也可以使用这样的方式进行计算。需要注意的是，这一次我们得到的结果不是行动本身，而是最优行动对于当前行动的偏差。由于偏差采用梯度的形式表示，因此在更新时我们可以进行多轮迭代更新，这样算法就变成了 iterative LQR。

13.2.3 iLQR 的实现

下面我们介绍 iLQR 的实现，这里参考 https://github.com/neka-nat/ilqr-gym 中的代码实现。这个项目主要实现了两部分功能。

1. 基于连续行动的平衡车环境

Gym 中实现了经典的控制任务 CartPole，环境中有一个小车，这个车只能左右移动，车的中心有一根钉子，这个钉子钉住了一个棍子的一头，对应的界面如图 13-7 所示。我们的目标是控制小车使棍子竖立在空中并保持一定时间，如果可以做到这一点，那么 Agent 就可以获得分数；如果棍子失去平衡掉落，那么 Agent 将无法获得分数。整个环境由一些物理规则实现，每一时刻 Agent 将获得 4 个状态观测值：小车的位置、小车的速度、棍子对应的角度，以及棍子的角速度。Agent 根据状态值给出对小车施加的力的大小。

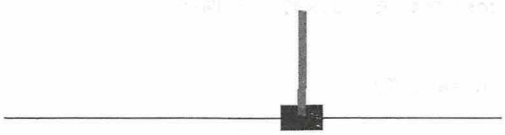


图 13-7 CartPole 界面图

2. iLQR 算法

这里我们直接将源代码展现出来，代码中同时包含对变量的介绍。首先是反向计算部分，这一部分比较复杂。

```
'''
代码变量介绍
v[t]: 代表V(x[t])
v_x[t]: 代表V(x[t])对于x的导数
v_xx[t]: 代表V(x[t])对于x的二阶导
```


书籍是人类进步的阶梯

```
f_x_t: 当前时刻f函数关于x的一阶偏导值
f_u_t: 当前时刻f函数关于u的一阶偏导值
q_x: 当前时刻q函数关于x的一阶偏导值
q_u: 当前时刻q函数关于u的一阶偏导值
q_xx: 当前时刻q函数关于x的二阶偏导值
q_uu: 当前时刻q函数关于u的二阶偏导值
q_ux: 当前时刻q函数关于u,x的二阶偏导值
kk, k: 代表计算最优u时的两个替代变量

代码函数介绍
costn(x): 代表最终状态的花费
costn_x(x): 代表最终状态花费对于x的导数
costn_xx(x): 代表最终状态花费对于x的二阶导
f_x(x, u): 代表f函数关于x的一阶偏导
f_u(x, u): 代表f函数关于u的一阶偏导
f_xx(x, u): 代表f函数关于x的二阶偏导
f_uu(x, u): 代表f函数关于u的二阶偏导
f_ux(x, u): 代表f函数关于u,x的二阶偏导
cost_x(x, u): 代表cost函数关于x的一阶偏导
cost_u(x, u): 代表cost函数关于x的一阶偏导
cost_xx(x, u): 代表cost函数关于x的二阶偏导
cost_uu(x, u): 代表cost函数关于x的二阶偏导
cost_ux(x, u): 代表cost函数关于u,x的二阶偏导
'''
def backward(x_seq, u_seq, T):
    v = np.zeros(T)
    v_x = np.zeros_like(v)
    v_xx = np.zeros([T, T])
    # 初始化最终状态的V函数
    v[-1] = costn(x_seq[-1])
    v_x[-1] = costn_x(x_seq[-1])
    v_xx[-1] = costn_xx(x_seq[-1])
    k_seq = []
    kk_seq = []
    for t in range(self.pred_time - 1, -1, -1):
        # step 1: 计算Q函数的对应部分
        f_x_t = self.f_x(x_seq[t], u_seq[t])
```


书籍是人类进步的阶梯

```
            f_u_t = self.f_u(x_seq[t], u_seq[t])
            q_x = self.cost_x(x_seq[t], u_seq[t]) + np.matmul(f_x_t.T,
                self.v_x[t + 1])
            q_u = self.cost_u(x_seq[t], u_seq[t]) + np.matmul(f_u_t.T,
                self.v_x[t + 1])
            q_xx = self.cost_xx(x_seq[t], u_seq[t]) + \
                np.matmul(np.matmul(f_x_t.T, self.v_xx[t + 1]), f_x_t) + \
                np.dot(self.v_x[t + 1], np.squeeze(self.f_xx(x_seq[t],
                u_seq[t])))
            tmp = np.matmul(f_u_t.T, self.v_xx[t + 1])
            q_uu = self.cost_uu(x_seq[t], u_seq[t]) + np.matmul(tmp, f_u_t)+ \
                np.dot(self.v_x[t + 1], np.squeeze(self.f_uu(x_seq[t],
                    u_seq[t])))
            q_ux = self.cost_ux(x_seq[t], u_seq[t]) + np.matmul(tmp, f_x_t)+ \
                np.dot(self.v_x[t + 1], np.squeeze(self.f_ux(x_seq[t],
                    u_seq[t])))
            # step 2: 计算最优u值
            inv_q_uu = np.linalg.inv(q_uu)
            k = -np.matmul(inv_q_uu, q_u)
            kk = -np.matmul(inv_q_uu, q_ux)
            # step 3: 计算V函数对应部分
            dv = 0.5 * np.matmul(q_u, k)
            self.v[t] += dv
            self.v_x[t] = q_x - np.matmul(np.matmul(q_u, inv_q_uu), q_ux)
            self.v_xx[t] = q_xx + np.matmul(q_ux.T, kk)
            k_seq.append(k)
            kk_seq.append(kk)
            k_seq.reverse()
            kk_seq.reverse()
        return k_seq, kk_seq
```

其次是前向计算。我们可以将反向计算求解出的参数带入前向计算中，这样就得到了更新后的行动：

```
def forward(self, x_seq, u_seq, k_seq, kk_seq):
    x_seq_hat = np.array(x_seq)
    u_seq_hat = np.array(u_seq)
    for t in range(len(u_seq)):
```


书籍是人类进步的阶梯

```
            control = k_seq[t] + np.matmul(kk_seq[t], (x_seq_hat[t] -
                x_seq[t]))
            u_seq_hat[t] = np.clip(u_seq[t] + control, -self.umax, self.umax)
            x_seq_hat[t + 1] = self.f(x_seq_hat[t], u_seq_hat[t])
        return x_seq_hat, u_seq_hat
```

经过一段时间的训练，模型就可以控制小车将棍子稳定地竖立在空中。感兴趣的读者可以亲自尝试。

LQR 属于微分动态规划（Differential Dynamic Programming，简称 DDP）中的一种算法，这样我们就能把第 6 章中的动态规划法和本节介绍的方法统一起来，两者是针对不同空间的动态规划方法。关于 DDP 的内容还有很多，其中还会涉及很多很复杂的内容，由于篇幅的原因这里不做深入介绍。

13.3 总结

本章我们主要介绍了模型已知问题的求解方法，总结如下。

（1）AlphaZero 采用了基于 MCTS 的自我学习方法进行策略学习。

（2）iLQR 通过依时间迭代计算的方式对控制类问题进行规划求解。

13.4 参考资料

[1] Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search[J]. Nature, 2016, 529(7587):484-489.

[2] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of Go without human knowledge[J]. Nature, 2017, 550(7676):354-359.

[3] Silver D, Hubert T, Schrittwieser J, et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm[J]. 2017.

[4] Browne C B, Powley E, Whitehouse D, et al. A Survey of Monte Carlo Tree Search Methods[J]. IEEE Transactions on Computational Intelligence & Ai in Games, 2012, 4:1(1):1-43.

[5] Tassa Y, Erez T, Todorov E. Synthesis and stabilization of complex behaviors through online trajectory optimization[C]// Ieee/rsj International Conference on Intelligent Robots and Systems. IEEE, 2012:4906-4913.

书籍是人类进步的阶梯

第五部分
反向强化学习

第五部分将介绍反向强化学习的知识。反向强化学习同样基于马尔可夫决策过程，不同的是过程中的回报是未知的，需要通过最优策略的信息推断出回报的形式。很多任务都存在回报形式不明，最优策略形式未知的问题，这些任务都可以通过强化学习与反向强化学习结合的方式求解。

第五部分共包含两章：

- 第 14 章将介绍反向强化学习的基础，其中包括问题的基本求解方法。
- 第 15 章将介绍更高级的算法，包括基于最大熵的反向强化学习（MEIRL）和生成对抗模仿学习（GAIL）。

书籍是人类进步的阶梯

14 反向强化学习入门

看名字就知道，反向强化学习和强化学习有很密切的关系。本章我们要解决的问题将变得更极端，在与环境交互的过程中，我们无法感知回报，但幸运的是，我们可以使用反向强化学习来解决这个问题。本章我们将介绍反向强化学习的基本概念及基本的解决方法，包括反向强化学习的定义和基本解法。虽然这些方法在实际问题中已经很少使用，但它是了解更高级方法的基础，值得我们深入理解。

14.1 基本概念

反向强化学习也被称为模仿学习（Imitation Learning）或学徒学习（Apprentice Learning），它是解决另一类强化学习问题的方法。既然如此，我们就回到强化学习的问题范畴中。在介绍强化学习的过程中，我们了解了它的基本结构：Agent、环境、状态、行动和回报。通过这些概念，我们将整个强化学习的基础搭建好，本书前面介绍的算法都在这个框架中执行。

那么这个框架究竟好不好呢？当然是好的，它已经帮助我们解决了很多实际问题。例如，很多基于强化学习的电子游戏机器人已经达到了游戏高手的水平；但是我们也必须看到，游戏是强化学习应用比较广泛的一个领域。那么我们不禁去思考一个问题：为什么强化学习在游戏方面斩获颇丰呢？

这就要回到问题的本源回报上来。在游戏设计的世界中，回报是一个极其关键的因素，游戏之所以吸引人，其中一个很重要的原因就是它会为玩家每一个小阶段的表

现赋予一个评价或者得分，这些得分可以很及时地给予玩家被量化的回报，而这些量化的回报也恰恰是强化学习算法十分需要的。

回到现实生活中，我们生活的真实世界——到目前为止，还无法用量化的回报清晰地表示一切行动。很多时候，我们需要人为地制定一些量化回报的规则，才能进行强化学习训练。但实际上，这些规则有时并不容易制定。

以游泳为例，人类是如何学习游泳的呢？如果我们报名游泳辅导班，老师会先带领我们进行监督学习——在岸上划臂蹬腿，然后直接指出我们的动作和标准动作的差距。随着我们不断地学习，我们的姿势也在不断提高。

而独立学习游泳时，我们就进入了另外一种学习模式。如果把独立学习游泳想象成强化学习，那么我们每完成一次游泳的动作都应该获得一定的反馈，但事实上，在真实的游泳过程中，这种反馈是很难获得的。我们还可以通过和小伙伴进行游泳比赛，知道谁游得快、谁游得慢，从而了解自己在游泳时的长处与不足，但是这个过程就像第12章介绍的问题一样，整个过程的反馈十分稀疏，甚至不足以为学习提供足够的信息。这样看来，独立学习游泳实在是一件事倍功半的事情。

那么反向强化学习能不能解决这个问题呢？反向强化学习的思路和强化学习相反，强化学习通常是在回报已知的情况下求出值函数和策略，但如果回报不容易得到，我们就得创造一些回报，但如果回报完全无法获得，学习就会困难很多。反向强化学习使用了逆向思维：我们不通过回报求策略，而是反过来，通过策略求回报。

这听上去有些荒谬，我们想通过交互获得回报，就是为了求解更好的策略；现在有了策略，还回去求回报干什么呢？当然，很多时候最优或者接近最优的策略是存在的，但是它不属于我们；或者说虽然我们无法具体表示出这种策略，但是最优策略的采样是可以得到的。回到游泳学习上，虽然我们不像游泳健将那样拥有超高的游泳“策略”，可以通过肌肉的控制力和爆发力完成完美的泳姿，但是我们可以拿起手中的拍摄装置，或者打开电视和视频网站，观看这些人游泳比赛的慢动作回放，从而掌握这个策略与环境的“交互数据”。这些交互数据就可以帮助我们提高游泳的能力。

实际上，很多人也是用这种方法学习游泳的：在家认真观看高手的动作要领，并铭记于心，最后回到泳池一试身手。实际上，这就是一个完整的从反向强化学习到强化学习的过程。首先通过最优策略的交互样本学习到回报函数——什么样的姿势是正确的，应该获得更高的回报；什么样的姿势不正确，应该获得较低的回报。接下来就可以利用学习到的回报函数进行尝试，使用强化学习的方法学习到策略。如果再进行一次反向强化学习的过程，通过分析高手的动作和自己动作的差异，我们就可以得到新的回报函数。新的回报函数将再次用于改进策略，这样不断迭代，就可以完成策略的学习。

将上面的过程整理出来，可以得到下面的流程：

（1）随机生成一个策略作为初始策略。

（2）通过比较“高手”的交互样本和自己交互样本的差别，学习得到回报函数。

（3）利用回报函数进行强化学习，提高自己策略的水平。

（4）如果两个策略的差别已经不大，就可以停止学习了，否则回到第（2）步。

以上是从通俗的角度对反向强化学习的介绍，下面将从数学视角出发，对这个问题进行深入分析。

14.2 从最优策略求解回报

14.2.1 求解回报的目标函数

14.1 节介绍了反向强化学习的基本概念，也介绍了它和强化学习的区别。对于一个强化学习的问题，如果状态和行动空间已知，状态转移概率确定，可以通过回报函数求出最优策略；而对于反向强化学习来说，则通过最优策略或最优策略的行动采样，求出回报函数。本节将介绍反向强化学习中最简单的问题：已知最优策略，求回报函数。

如果我们已知了状态和行动空间、状态转移概率和最优策略 π^*，且状态和行动的空间都是离散的，那么我们该如何求解呢？我们使用表格的形式表示策略与值函数，这种表示形式在第 6 章和第 7 章已经介绍过。根据最优策略的定义我们知道，最优策略 π^* 和其他策略 π 相比，对于每一个状态，其对应的值函数都应不弱于其他策略的值函数。由此可以得到

$$V_{\pi^*}(\boldsymbol{s}) \geqslant V_{\pi}(\boldsymbol{s})$$
$$q_{\pi^*}(\boldsymbol{s}, \boldsymbol{a}) \geqslant q_{\pi}(\boldsymbol{s}, \boldsymbol{a})$$

如果令 $\boldsymbol{S}$ 为所有的状态集合，$\boldsymbol{A}$ 为所有的行动集合，那么对于任意的状态，可以得到

$$\forall \boldsymbol{s} \in \boldsymbol{S}, q_{\pi^*}(\boldsymbol{s}, \pi^*(\boldsymbol{s})) \geqslant q_{\pi}(\boldsymbol{s}, \pi(\boldsymbol{s}))$$

我们知道最优策略的每一步行动都应该不弱于其他策略的行动，那么可以得到对于任意的状态，完全使用最优策略的价值，应该不弱于“在当前时刻使用最优策略外的行动，此后使用最优策略进行决策”的价值。这个条件可以通过反证法得到，如果这一条件不能满足，那么实际上最优策略就不是最优了。上面的条件可以写作：

$$\forall \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{a} \in \boldsymbol{A} - \pi^*(\boldsymbol{s}),\ q_{\pi^*}(\boldsymbol{s}, \pi^*(\boldsymbol{s})) \geqslant q_{\pi^*}(\boldsymbol{s}, \boldsymbol{a})$$

我们可以利用 Bellman 公式将上面的不等式展开，得到

$$\forall \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{a} \in \boldsymbol{A} - \pi^*(\boldsymbol{s}),\ \sum_{\boldsymbol{s}'} P(\boldsymbol{s}'|\boldsymbol{s}, \pi^*(\boldsymbol{s}))V_{\pi^*}(\boldsymbol{s}') \geqslant \sum_{\boldsymbol{s}'} P(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})V_{\pi^*}(\boldsymbol{s}')$$

如果用列向量 $\boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}}$ 表示由 $P(\boldsymbol{s}'|\boldsymbol{s}, \pi^*(\boldsymbol{s}))$ 组成的向量，用列向量 $\boldsymbol{V}$ 表示由 $V_{\pi^*}(\boldsymbol{s}')$ 组成的向量，那么公式就可以变为

$$P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}\boldsymbol{V} \geqslant \boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}}\boldsymbol{V}$$

这个公式里的值函数 V 是未知的，但好在可以用 Bellman 公式进行求解：

$$V_{\pi^*}(\boldsymbol{s}) = \sum_{\boldsymbol{a}} P(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} P(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})[R(\boldsymbol{s}) + \gamma V_{\pi^*}(\boldsymbol{s}')]$$

将这个公式转成矩阵和向量计算的形式，用 π 表示行动概率向量 $P(\boldsymbol{a}|\boldsymbol{s})$，就可以得到

$$\boldsymbol{V} = \boldsymbol{R} + \pi P(\gamma \boldsymbol{V})$$

由于我们在考虑表格式策略的值函数，所以策略是确定的，公式可以进一步化简为

$$\boldsymbol{V} = \boldsymbol{R} + P_{:,\pi^*}(\gamma \boldsymbol{V})$$

当值函数收敛时，公式中的两个价值向量是相等的，于是公式可以整理为关于价值向量的形式：

$$\boldsymbol{V} = (\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^*})^{-1}\boldsymbol{R}$$

把这个公式带回前面的不等式中，就得到了我们初步的目标函数：

$$\begin{aligned} &\max(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})} - \boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}})(\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^*})\boldsymbol{R} \\ &\text{s.t.} \quad (P_{\boldsymbol{s},\pi^*(\boldsymbol{s})} - \boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}})(\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^*})\boldsymbol{R} \geqslant 0 \end{aligned}$$

于是这个问题变成了一个线性规划问题，我们要找到合适的回报向量满足这个不等式。经过上面的推导，我们终于完成了问题定义的第一步，通过最优策略的定义和 Bellman 公式，我们将其表达出来。

14.2.2 目标函数的约束

虽然上面的目标函数清晰地阐述了求解的思路，但由于这个目标只是一个不等式，对待求的回报向量来说，这个约束有点弱。显然我们可以找出很多种回报函数组合满足这个不等式，其中还有一些是平凡解，例如全部为 0 回报函数，这样的解同样可以满足不等式，但实际上它是没有意义的。于是我们需要思考如何增强问题的约束。

我们将为目标函数增加两个约束：1）限制回报的范围；2）只考虑最优策略和次优策略的差异。

首先是限制回报的范围。由于这是一个线性规划问题，回报向量可以在成比例扩大的同时满足约束条件，因此我们应该对回报数值的上界做一定的限制，于是就有了这样的约束：

$$|\boldsymbol{R}| \leqslant \boldsymbol{R}_{\max}$$

这种增加约束的方法在机器学习中普遍存在，这和 L1、L2 正则的功能比较相似，都是为了防止参数太大。由于这种方法在机器学习的很多算法上都得到了不错的效果，所以它在这里的效果应该也不会差。

其次考虑最优策略的比较对象。在前面的目标不等式中，我们需要最优策略强于所有的策略，这个约束并没有错，但在实际中，如果我们以相同的权重考量其他策略和最优策略的差距，就显得不那么合理了。实际上，最优策略和次优策略的价值差会显得更重要，而最优策略和其他很差的策略的价值差并没那么重要，所以实际上最优策略只要尽可能地比次优策略强就可以了，和其他的策略进行比较并不应该拥有很高的权重。

我们以一个棋类 AI 的表现为例解释价值差异的权重问题。假设有两个策略，分别为 A 和 B，它们分别与普通玩家和高级玩家对弈，最终的战绩如表 14-1 所示。

表 14-1　策略 A 和 B 分别与普通玩家和高级玩家对弈的战绩

	普通玩家	高级玩家	总得分
A	20	0	20
B	2	2	4

这里的分数表示了两者的差距，分数为正表示 AI 最终胜利。可以看出，策略 A 获得了更高的分数，却和高级玩家打平；策略 B 的总得分虽然不及策略 A，但是它获得了两场胜利，这说明策略 B 与策略 A 相比，更像一个最优策略，即使它并不能从普通玩家手上获得太多的分数。

根据上面的分析，我们将目标函数变为

$$\max_{\boldsymbol{R}}[\min_{\boldsymbol{a}}\{(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}-\boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}})(\boldsymbol{I}-\gamma\boldsymbol{P}_{\pi^*})^{-1}\boldsymbol{R}\}]$$

从公式中可以看出，我们先通过内层的 min 运算选出与最优策略差距最小的策略，再通过外层的 max 运算求解 Reward 使最优策略与它的差距变大，这样 Reward 的求解就变得更明确了。

将两个约束加入原问题，此时完整的问题就变为

$$\begin{aligned}&\max_{\boldsymbol{R}}[\min_{\boldsymbol{a}}\{(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}-\boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}})(\boldsymbol{I}-\gamma\boldsymbol{P}_{\pi^*})^{-1}\boldsymbol{R}\}-\lambda|\boldsymbol{R}|]\\&\text{s.t.}\quad -(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}-\boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}})(\boldsymbol{I}-\gamma\boldsymbol{P}_{\pi^*})\boldsymbol{R}\leqslant 0\\&\qquad |\boldsymbol{R}|\leqslant \boldsymbol{R}_{\max}\end{aligned}$$

这样我们就得到了这个问题的完整目标，14.3 节将介绍这个问题的具体求解方式。

14.3 求解线性规划

14.3.1 线性规划的求解过程

14.2 节介绍了反向强化学习的公式形式，接下来完成求解。这个问题本质上是一个线性规划的问题，但目前的公式形式还不是一个易于求解的形式，于是我们要将公式转变，采用变量替换的方式，将问题变换成标准的线性规划形式。这里涉及两个变量替换：

$$\begin{aligned}&\boldsymbol{t}\leftarrow \min_{\boldsymbol{a}}\{(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}-\boldsymbol{P}_{\boldsymbol{s},\boldsymbol{a}})(\boldsymbol{I}-\gamma\boldsymbol{P}_{\pi^*})^{-1}\boldsymbol{R}\}\\&\boldsymbol{u}\leftarrow|\boldsymbol{R}|\end{aligned}$$

于是可以得到

$$\begin{aligned}&\max\{\sum_{i=1}^{|\boldsymbol{S}|}\{\boldsymbol{t}_i-\lambda\boldsymbol{u}_i\}\}\\&\text{s.t.}\quad (P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}(i)-P_{\boldsymbol{s},\boldsymbol{a}}(i))(\boldsymbol{I}-\gamma\boldsymbol{P}_{\pi^*})^{-1}\boldsymbol{R}\geqslant \boldsymbol{t}_i,\\&\qquad \forall\boldsymbol{a}\in\boldsymbol{A}-\pi^*(\boldsymbol{s}),i=1,\ldots|\boldsymbol{S}|(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}(i)-P_{\boldsymbol{s},\boldsymbol{a}}(i))(\boldsymbol{I}-\gamma\boldsymbol{P}_{\pi^*})^{-1}\boldsymbol{R}\geqslant 0,\end{aligned}$$

$$\forall \boldsymbol{a} \in \boldsymbol{A} - \pi^*(\boldsymbol{s}), i = 1, \ldots, |\boldsymbol{S}|$$
$$-\boldsymbol{u} \leqslant \boldsymbol{R} \leqslant \boldsymbol{u} |\boldsymbol{R}_i| \leqslant \boldsymbol{R}_{\max}$$

当我们将公式变成这个形式后，就可以进行求解了。我们可以从项目 https://github.com/MatthewJA/Inverse-Reinforcement-Learning 中找到关于这个问题的求解，具体的实现在 irl/linear_irl.py 中的 irl 函数，下面笔者就来介绍它是如何进行求解的。

代码中使用了 cvxopt 这个库，它用来实现常见的凸优化算法，其中也包括线性规划的求解方法。想要使用这个算法库进行求解，就需要将问题整理成下面的形式：

$$\min \boldsymbol{c}^{\mathrm{T}} \boldsymbol{x}$$
$$\text{s.t.} \quad \boldsymbol{A}\boldsymbol{x} \leqslant \boldsymbol{b}$$

其中 $\boldsymbol{A}$ 为矩阵，$\boldsymbol{b}$、$\boldsymbol{c}$ 为向量。最终我们将 $\boldsymbol{A}$、$\boldsymbol{b}$、$\boldsymbol{c}$ 三部分输入函数中，就可以利用算法库中的线性规划算法进行求解。

那么，上面的公式该怎么变成这个样子呢？为了方便计算，此时我们共有三组变量 $\boldsymbol{R}$、$\boldsymbol{T}$ 和 $\boldsymbol{U}$，三组变量的长度都是 $|\boldsymbol{S}|$，所以我们的目标函数变为

$$\min -[0 \ldots 0, 1 \ldots 1, -\lambda \ldots -\lambda]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}]$$

接下来按照类似的形式，我们得到下面的条件约束：

$$[-(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}(i) - P_{\boldsymbol{s},\boldsymbol{a}}(i))(\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^*})^{-1}, -1, 0]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}] \leqslant 0$$
$$[-(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}(i) - P_{\boldsymbol{s},\boldsymbol{a}}(i))(\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^*})^{-1}, 0, 0]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}] \leqslant 0$$
$$[-1, 0, -1]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}] \leqslant 0$$
$$[1, 0, -1]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}] \leqslant 0$$
$$[-1, 0, 0]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}] \leqslant \boldsymbol{R}_{\max}$$
$$[1, 0, 0]^{\mathrm{T}} [\boldsymbol{R}, \boldsymbol{T}, \boldsymbol{U}] \leqslant \boldsymbol{R}_{\max}$$

上面的公式无法详细展示计算过程，于是我们画出了问题的具体计算形式和每一个计算量的维度，如图 14-1 所示，其中 $T(\boldsymbol{a}, \boldsymbol{s})$ 表示公式 $(P_{\boldsymbol{s},\pi^*(\boldsymbol{s})}(i) - P_{\boldsymbol{s},\boldsymbol{a}}(i))(\boldsymbol{I} - \gamma \boldsymbol{P}_{\pi^*})^{-1} \boldsymbol{R}$ 的计算值，对于每一个状态，我们都要让最优策略的行动和其他行动进行比较，因此这个公式维度为 $|\boldsymbol{S}| \times (|\boldsymbol{A}| - 1)$。

按照图 14-1 所示将数据组织好后，就可以求解得到回报向量了。

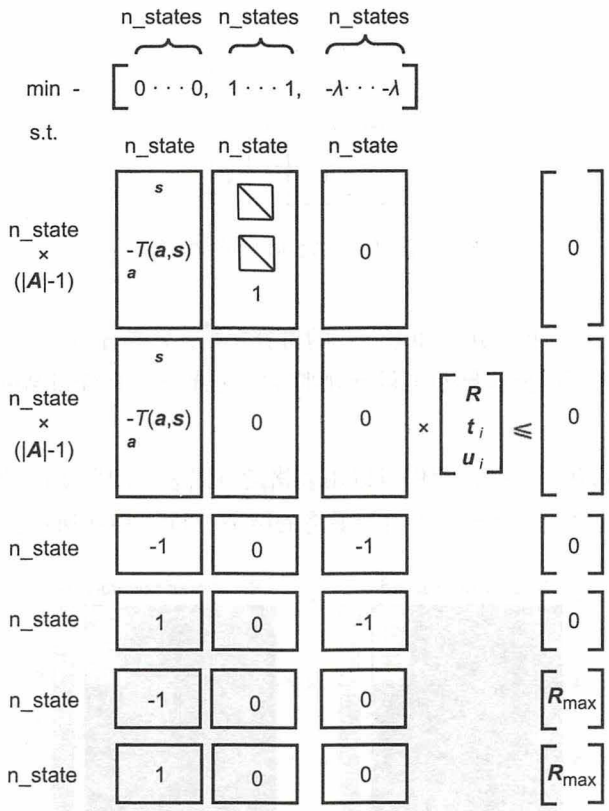


图 14-1 线性规划运算的数据结构

14.3.2 实际案例

看完了上面的算法，我们再来看一个具体的例子，这个例子是 Grid World。Grid World 是一个很简单的问题，我们给定一个 $N \times N$ 的棋盘格，这个棋盘格上的每一个格子都可以作为落脚点。我们以 5×5 的格子为例，如图 14-2 所示。

站在棋盘的格子上会产生不同的效果，棋盘格的右下角是一个会得到奖励的格子，站在上面会获得 1 分，游戏结束；站在其他的格子上不会获得奖励，游戏仍会继续。

玩家在初始状态下会随机站在某个格子上，接下来的每一个时刻，玩家都可以做出决定，选择上下左右 4 个方向行进。世事无常，有时玩家并不能到达想去的地方，游戏中有一股“妖风”，它会以一定的概率触发。触发后玩家的行进将不受控制，会任意选择一个方向前进，如果撞到棋盘的边缘，玩家将被阻挡在那里，停止前进。

整个游戏的规则并不复杂，从在玩家的角度看，因为右下角是得分点，因此只要不停地向那个方向前进就可以了；如果已经到达了那里，那么保持停留就可以。

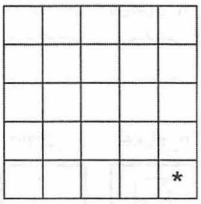


图 14-2　Grid World 的游戏形式

我们换一个思路：我们并不知道游戏的得分规则，只能看到一个老玩家的操作。我们能不能通过老玩家的操作推断出得分规则呢？这就是基于 Grid World 的反向强化学习问题。

由于这个问题并不复杂，我们可以用表格的方式表示问题中的策略、回报和价值向量，运用上面的算法，我们可以得到最终的结果，如图 14-3 所示。

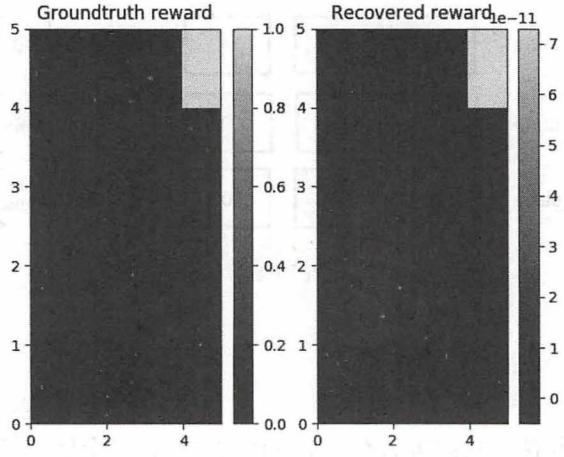


图 14-3　经过计算得到的 Grid World 回报图

从结果可以看出，我们复原的这个 5 × 5 的 Reward 结果和真实结果相差很小，或者说这点差距并不能阻碍我们选出正确的策略。以上就是反向强化学习的基本解法。

14.4　无限状态下的求解

14.3 节介绍了反向强化学习的基本解法，但是它所应用的场景比较有限，以状态空间为例，我们需要状态空间中状态的数量是有限的，行动的数量也是有限的。遇到无限状态空间的问题时，该如何确定回报函数呢？

首先要解决的是回报的表示问题。回报函数是一个从状态到回报的映射函数，在14.3节的问题中，由于状态数量是有限的，所以这个函数可以以向量的形式表示出来；但是现在状态数量不再是有限的，显然不能继续用表格的形式表示，我们必须做出改变。

为了解决这个问题，我们可以定义一个映射函数 $\phi(\boldsymbol{s})$，它可以把状态映射到一个低维空间，再进行计算。映射函数的具体形式可以根据问题的具体情况定义，也可以把这部分想象成特征工程要完成的工作：将无限维的状态用有限维的特征表示出来，这样回报函数可以用一个线性函数表示，这个函数的形式为

$$\text{Reward}(\boldsymbol{s}) = w^{\mathrm{T}}\phi(\boldsymbol{s})$$

于是我们的问题就变成了通过最优策略求解参数 w 来确定回报函数。

读者可能会有一个疑问，为什么要把这个函数定义为一个线性函数，而不是其他的函数呢？当然是有原因的：这个函数的形式对后面的计算具有一定的简便性。我们可以将上面的公式展开，假设每一个状态被映射到了一个 d 维的向量上，那么求解回报就相当于进行如下计算：

$$R(\boldsymbol{s}) = w_1\phi_1(\boldsymbol{s}) + w_2\phi_2(\boldsymbol{s}) + \cdots + w_d\phi_d(\boldsymbol{s})$$

我们又知道值函数的计算方式为

$$V^{\pi}(\boldsymbol{s}) = R^t(\boldsymbol{s}) + \gamma R^{t+1}(\boldsymbol{s}) + \gamma^2 R^{t+2}(\boldsymbol{s}) + \ldots = \sum_{t=0}^{\infty}\gamma^t R^t(\boldsymbol{s})$$

将两个公式融合，可以得到

$$\begin{aligned} V^{\pi}(\boldsymbol{s}) &= \sum_{t=0}^{\infty}\gamma^t \sum_{i=1}^{d} w_i\phi_i(\boldsymbol{s}) \\ &= \sum_{i=1}^{d}\sum_{t=0}^{\infty}\gamma^t w_i\phi_i(\boldsymbol{s}) \\ &= \sum_{i=1}^{d} w_i \sum_{t=0}^{\infty}\gamma^t\phi_i(\boldsymbol{s}) \\ &= \sum_{i=1}^{d} w_i V_i^{\pi}(\boldsymbol{s}) \end{aligned}$$

根据上面的变换，我们发现值函数被整理成了一个比较简单的形式，其中 $V_i^{\pi}(\boldsymbol{s})$

表示某一维映射特征的累积值。我们最终的目标同样是希望找到最佳的回报函数，使最优策略的价值最大化，也就是说：

$$E_{s'\sim P_{s,\pi^*(s)}}[V^{\pi}(s')] \geqslant E_{s'\sim P_{s,a}}[V^{\pi}(s')]$$

$$\sum_{i=1}^{d} w_i(E_{s'\sim P_{s,\pi^*(s)}}[V_i^{\pi}(s')] - E_{s'\sim P_{s,a}}[V_i^{\pi}(s')]) \geqslant 0$$

最终的求解目标为

$$\text{maximize} \sum_{s\in S_0} \min_{a\in\{a_2,\ldots a_k\}}\{p(\sum_{i=1}^{d} w_i(E_{s'\sim P_{s,\pi^*(s)}}[V_i^{\pi}(s')] - E_{s'\sim P_{sa}}[V_i^{\pi}(s')]))\}$$

$$\text{s.t.} \quad |w_i| \leqslant 1, i = 1, \ldots, d$$

这个目标函数与 14.3 节的目标函数有 3 个不同。

第 1 个不同是状态的数量，由于状态空间变得非常大，我们无法将所有的状态纳入考虑范畴，因此只能考虑有限个状态，在公式中用 S_0 表示所有参与优化的状态，这是全部状态的子集。

第 2 个不同是对回报的约束。在 14.3 节的算法中，我们直接对回报的值进行设定，而对本节的目标函数来说，我们再对回报值做设定将变得有些困难，因此这里直接对回报函数的参数的大小进行约束，这样约束就更像 L1 范数约束了。

第 3 个不同是由于函数形式变换带来的修改。第 7 章介绍过表格式模型和函数式模型的特点，这些特点在这里同样适用。表格式模型虽然在一些问题上难以使用，但它可以完美地拟合真实的回报函数；而函数式模型由于可变量有限，并不能保证完美拟合真实的回报函数。一旦无法完美拟合，函数式模型就有可能出现违背约束条件的情况，即可能出现“最优策略”不是最优的情况。为了解决这个问题，我们需要将约束项转变为惩罚项，对模型中违反约束条件的部分进行高权重的惩罚。这里我们使用了惩罚函数 $p(x)$，它的形式为

$$p(x) = \begin{cases} x & \text{如果 } x \geqslant 0 \\ 2x & \text{如果 } x < 0 \end{cases}$$

接下来，我们就要对这个问题进行求解。从目标公式上看，这个问题和上面的问题形式差不多：同为线性规划，可以用同样的方法求解。我们可以从项目 https://github.com/MatthewJA/Inverse-Reinforcement-Learning 中找到关于这个问题的求解，具体的实现在 irl/linear_irl.py 中的 large_irl 函数，下面笔者就来介绍它是如何进行求解的。

为了将目标函数转变成优化库可以求解的形式，我们要先解决 $p(\text{gap}_{s,a})$ 这个函数的表示问题，其中 $\text{gap}_{s,a}=\sum_{i=1}^{d} w_i(\boldsymbol{E}_{s\sim P_{s,\pi^*(s)}}[V_i^{\pi}(s)]-E_{s\sim P_{s,a}}[V_i^{\pi}(s)])$。它包含一个分支条件，因为我们有 $|\boldsymbol{S}_0|$ 个状态，$|\boldsymbol{A}|-1$ 个非最优策略的行动，共有 $|\boldsymbol{S}_0|\times(|\boldsymbol{A}|-1)$ 个状态价值比较对，于是我们就需要设立 $2|\boldsymbol{S}_0|\times(|\boldsymbol{A}|-1)$ 个变量表示这些比较对可能的绝对值。我们令 $\text{gap}_{s,a}^{+}$ 表示 $\text{gap}_{s,a}$ 为正时的绝对值，也就是说 $\text{gap}_{s,a}^{+}=\text{gap}_{s,a}$，$\text{gap}_{s,a}^{-}$ 表示 $\text{gap}_{s,a}$ 为负的绝对值，即 $\text{gap}_{s,a}^{-}=-\text{gap}_{s,a}$，这样就产生了下面五个约束：

$$
\begin{aligned}
&-\text{gap}_{s,a}^{+}+\text{gap}_{s,a}^{-}+\text{gap}_{s,a}=0\\
&-\text{gap}_{s,a}^{-}\leqslant 0\\
&-\text{gap}_{s,a}^{+}\leqslant 0\\
&p(\text{gap}_{s,a}^{+})=\text{gap}_{s,a}^{+}\\
&p(-\text{gap}_{s,a}^{-})=-2\text{gap}_{s,a}^{-}
\end{aligned}
$$

完成这一步的变换，下一步要做的就是描述最优比次优好的条件，我们用另一个变量 u_s 表示 $\min_{a\in\{a_2,\cdots,a_k\}}\{p(\text{gap}_{s,a})\}$，根据最小值的约束，可以得到

$$
u_s\leqslant p(\text{gap}_{s,a})=p(-\text{gap}_{s,a}^{-}+\text{gap}_{s,a}^{+})
$$

由于前面约束的存在，我们可以将其拆成两个约束：

$$
\begin{aligned}
&u_s\leqslant p(-\text{gap}_{s,a}^{-})+p(\text{gap}_{s,a}^{+})\\
&u_s\leqslant -2\text{gap}_{s,a}^{-}+\text{gap}_{s,a}^{+}\\
&u_s-\text{gap}_{s,a}^{+}+2\text{gap}_{s,a}^{-}\leqslant 0
\end{aligned}
$$

最后我们的目标就是最大化每一个 u_s，所以完整的线性规划问题变为

$$
\begin{aligned}
&\text{Min}-\sum_{s\in\boldsymbol{S}_0}u_s\\
&\text{s.t.}\quad u_s-\text{gap}_{s,a}^{+}+2\text{gap}_{s,a}^{-}\leqslant 0,\forall s,\forall a\in\boldsymbol{A}-\pi(s)\\
&\qquad -\text{gap}_{s,a}^{-}\leqslant 0,\forall s,\forall a\in\boldsymbol{A}-\pi(s)\\
&\qquad -\text{gap}_{s,a}^{+}\leqslant 0,\forall s,\forall a\in\boldsymbol{A}-\pi(s)\\
&\qquad -w_d\leqslant 1,\forall d\\
&\qquad w_d\leqslant 1,\forall d
\end{aligned}
$$

$$-\text{gap}^{+}_{s,a}+\text{gap}^{-}_{s,a}+\text{gap}_{s,a}=0,\forall s,\forall a\in A-\pi(s)$$

得到了这个完整的公式后，我们就可以进行进一步地计算了。代码的作者在函数中实际上计算了 $\text{gap}_{s,a}$ 的相反数，但约束条件与上面的推导类似，请读者自行分析其中的不同。

14.5 从样本中学习

前面几节提到了两种反向强化学习方法，虽然我们已经将问题的规模扩展到了无限维的状态空间，但这和我们的真实情况还有一定的差距。毕竟在问题中，我们已经知道了最优策略的细节，而在很多情况下，我们最终的目标是求解最优策略，也就是说，最优策略是未知的。

本节将不再直接获得问题的最优策略，而是获得最优策略的交互样本。就像笔者提到过的：我们并不知道最佳的游泳姿势策略是怎样的，但是我们可以获得最佳游泳姿势的样本——世界冠军的游泳姿势，而这才是我们最常见的问题形式。所以从本节起，我们才真正开始了对真实反向强化学习问题的探索。

对于每一个最优策略的采样序列，我们可以将其列为

$$D=\{s_0,s_1,\dots s_n\}$$

我们可以设定它是有限的。通过对序列进行值函数的计算，我们得到了这个序列的值函数：

$$V(s_0)=R(s_0)+\gamma R(s_1)+\gamma^2R(s_2)+\cdots+\gamma^nR(s_n)$$

如果有某个策略函数 π，从状态 s_0 开始，令策略 π 和环境进行交互，最终产生了如下状态序列：

$$D'=\{s_0,s'_1,\dots,s'_n\}$$

这个序列的值函数为

$$V(s'_0)=R(s'_0)+\gamma R(s'_1)+\gamma^2R(s'_2)+\cdots+\gamma^nR(s'_n)$$

得到了两个序列的值函数，就可以使用和前面章节一样的目标：最优策略的价值不低于其他策略。由于我们只有策略 π，于是就让策略 π 作为“其他策略”的代表。我

们令最优采样的序列为 $\boldsymbol{D}$，那么根据最优策略的定义，可以得到如下目标函数：

$$\max_w(\sum_i w_i[V^i_{s\sim \boldsymbol{D}}(s_0) - V^i_{s\sim\pi}(s_0)])$$

看到这个目标函数，相信读者已经有点熟悉了。在 14.4 节中，我们也得到了类似的目标函数，使用了一个映射函数，将无限维的状态映射到有限维的空间中。当然，这个公式和 14.4 节的形式存在一些不同。在 14.4 节中，我们需要让最优策略尽可能地大于次优策略，但是现在我们无法找到次优策略，只能使用策略 π。

14.1 节介绍了反向强化学习和强化学习结合的算法，也列出了算法的详细步骤。我们从一个随机采样得到的策略出发，让这个策略从最优策略采样的起始状态出发进行行动决策，我们用这个随机策略得到了一系列行动轨迹；有了行动轨迹，就可以用反向强化学习的模型求解回报函数了；当我们得到了回报函数，就可以利用强化学习对现有的策略进行改进，前面的章节已经了解了很多改进的方法，这里不再赘述。策略完成改进，就可以以这个策略重新完成一轮优化，也就是再进行一次采样、回报函数优化和强化学习改进。这样一轮一轮地不断改进，就可以让策略的效果不断提升。随着策略的不断提升，它产生的序列的回报总和会越来越靠近最优策略轨迹的回报，策略模型也会越来越接近“次优策略”，甚至超过次优策略，因此对应的回报函数也会估计得更准确，这个过程可以用图 14-4 表示。

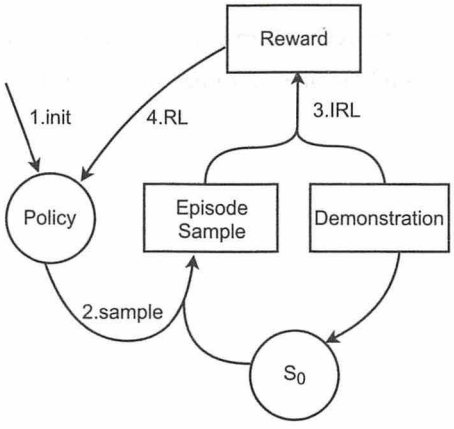


图 14-4 基于序列样本的强化学习流程

最终，我们得到了两个结果。

（1）精确的回报函数，它使得最优策略采样不弱于此前的所有策略函数。

（2）一个无限接近最优策略的策略函数，经过若干轮的迭代优化，它的效果不断增强，直至无法提高。

仅供非商业用途或交流学习使用

以上就是针对这类问题的解决方案，更多反向强化学习的算法将会在第 15 章介绍。

14.6 总结

本章介绍了反向强化学习的基本知识，让我们总结一下。

（1）反向强化学习主要用于求解环境回报的问题，为了求解回报，我们需要知道最优策略或者最优策略的交互序列。

（2）为了学习状态回报，我们需要确保最优策略下的状态价值不小于其他策略。

（3）为了使求解出的回报更有意义，我们需要限定回报的上限，同时关注最优策略和次优策略在状态价值上的关系。

（4）当需要基于最优序列样本学习策略时，我们可以结合反向强化学习和强化学习，共同提高回报函数的精确度和策略的效果。

14.7 参考资料

[1] Ng A Y, Russell S J. Algorithms for Inverse Reinforcement Learning[C]// International Conference on Machine Learning. 2000:663-670.

[2] Abbeel P, Ng A Y. Apprenticeship learning via inverse reinforcement learning[C]// International Conference on Machine Learning. ACM, 2004:1.

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

15 反向强化学习算法 2.0

第 14 章是对反向强化学习的基础进行介绍，相信读者对反向强化学习已经有了一定的了解。第 14 章介绍的解法相对简单，可以解决小规模的问题，而解决大规模的问题时就有些吃力了。本章的目标是学习更好的反向强化学习算法，它们可以解决更复杂的问题。这些方法并不是这个领域独有的，而是在其他机器学习领域得到了一定的验证，获得了一定的成功后，才被应用到这个领域上的。由于“站在巨人的肩膀上”，这些方法的效果也得到了肯定。首先要介绍的是基于最大熵的反向强化学习方法（Max Entropy Inverse Reinforcement Learning），它能够从多个满足限定条件的策略中寻找一个更合理的策略。由于最大熵的算法有一点复杂，因此我们要先对它的基本原理进行介绍。然后介绍基于生成对抗的模仿学习（Generative Adversarial Imitation Learning），它使用了近年来大获成功的生成对抗模型的思想和模型结构，通过挖掘它和反向强化学习的相似性实现了类似的方法。

15.1 最大熵模型

想要了解基于最大熵的反向强化学习，先要了解最大熵模型。实际上，最大熵的思想是一个很常见、也很符合直觉的算法。为了更深刻地理解最大熵模型，我们还要介绍一个和最大熵模型相伴的概念——指数家族。指数家族是一个包含我们常见的概率分布的分布族，当我们理解了这个神奇的家族后，最大熵也就变得不那么神秘了。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

15.1.1 指数家族

第 2 章介绍了一些概率分布模型，其中包括离散概率分布的代表伯努利分布和连续概率分布的代表高斯分布。虽然这两个分布的形式不同，但它们存在着一定的相似性，那就是——它们都属于指数家族。这个家族是一个经典且独具特色的家族，其中包含了很多我们常见的模型。这个家族中的分布拥有一些很有趣的性质，这些性质有时可以帮助我们解决一些大问题。

首先我们来看看这个大家族的样子。既然它可以表示很多概率分布，那么它一定拥有一个十分抽象的形式。这个形式抽象到让我们很难将它和它的成员联想到一起的地步。指数家族的基本公式形式为

$$p(x|\theta) = h(x)\exp(\theta^{\mathrm{T}}\phi(x) - A(\theta))$$

公式中的每一项都有自己的名字。让我们从左往右依次介绍：$h(x)$ 一般被认为是一个简单的乘子项，很多时候它只是一个常量。θ 表示模型中的参数，$\phi(x)$ 被称为函数的充分统计量（Sufficient Statistic），$A(\theta)$ 被称为对数分割函数（Log Partition Function），一般被用来归一化所有的概率项，它的具体形式为

$$A(\theta) = \log \int_{x \in X} h(x)\exp(\theta^{\mathrm{T}}\phi(x))\mathrm{d}x$$

读者一定觉得这个形式十分陌生，它怎么会和我们常见的模型产生联系呢？这个公式如何与我们常见的模型对应呢？我们就以上面的两个分布为例，通过变换找出它们与指数分布的联系。首先是伯努利分布，它的形式为

$$P(X = x) = p^x(1-p)^{1-x}$$

其中 p 表示 $X = 1$ 时的概率。将这个公式转变，可以变成类似指数分布的形式：

$$\begin{aligned} P(X = x) &= \exp(\log(p^x(1-p)^{1-x})) \\ &= \exp(x\log p + (1-x)\log(1-p)) \\ &= \exp(x(\log p - \log(1-p)) + \log(1-p)) \\ &= \exp(x\log\frac{p}{1-p} - \log\frac{1}{1-p}) \end{aligned}$$

和指数分布的形式对比，我们发现其中的每一项可以做如下替换：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$
\begin{aligned}
h(x) &= 1 \\
\phi(x) &= x \\
\theta &= \log(\frac{p}{1-p}) \\
A(\theta) &= \log\frac{1}{1-p} \\
&= \log(1+\frac{p}{1-p}) \\
&= \log(1+\exp(\theta))
\end{aligned}
$$

这样我们可以用指数家族的标准形式表示伯努利分布。虽然看上去这样的表示有些复杂，但是它们在形式上还是完成了统一。同样地，高斯分布也可以变换成类似的形式：

$$
\begin{aligned}
N(x|\mu,\sigma^2) &= \frac{1}{\sqrt{2\pi\sigma^2}}\exp[-\frac{(x-\mu)^2}{2\sigma^2}] \\
&= \frac{1}{\sqrt{2\pi}}\exp[-\log\sigma-\frac{(x-\mu)^2}{2\sigma^2}] \\
&= \frac{1}{\sqrt{2\pi}}\exp[-\log\sigma-\frac{1}{2\sigma^2}((x^2-2x\mu+\mu^2))] \\
&= \frac{1}{\sqrt{2\pi}}\exp[-\log\sigma-\frac{1}{2\sigma^2}\mu^2+\frac{1}{\sigma^2}x\mu-\frac{1}{2\sigma^2}x^2]
\end{aligned}
$$

和指数分布的形式对比，我们发现其中的每一项可以做如下替换：

$$
\begin{aligned}
h(x) &= \frac{1}{\sqrt{2\pi}} \\
\theta &= [\frac{\mu}{\sigma^2},-\frac{1}{2\sigma^2}] \\
\phi(x) &= [x,x^2] \\
A(\theta) &= \log\sigma+\frac{1}{2\sigma^2}\mu^2 \\
&= \log\sqrt{\sigma^2}+(\frac{\mu}{\sigma^2})^2\times\frac{1}{2}\sigma^2 \\
&= -\log\sqrt{\frac{1}{\sigma^2}}+\theta_1^2\times\frac{1}{2\times\frac{1}{\sigma^2}} \\
&= -\log\sqrt{-2\times-\frac{1}{2\sigma^2}}-\theta_1^2\times\frac{1}{4\times-\frac{1}{2\sigma^2}} \\
&= -\log\sqrt{-2\times\theta_2}-\theta_1^2\times\frac{1}{4\times\theta_2}
\end{aligned}
$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$= -\frac{1}{2}\log(-2\theta_2) - \frac{\theta_1^2}{4\theta_2}$$

我们发现，本身有些复杂的高斯分布经过变换变成了更复杂的指数家族通用形式。这时读者一定会感到疑惑：为什么要将这些分布变换成这样的形式呢？因为指数分布具有一些非常好的性质，有时它们可以帮我们解决大问题。下面就来看两个简单的性质。

第一个性质与对数分割函数有关：$A(\theta)$ 的一阶导等于 $\phi(x)$ 的期望。听上去有些不可思议，让我们对这个性质进行证明，可以得到

$$\begin{aligned}
\frac{\partial A(\theta)}{\partial \theta} &= \frac{\partial}{\partial \theta}(\log \int h(x)\exp\{\theta^{\mathrm{T}}\phi(x)\}\mathrm{d}x) \\
&= \frac{\int h(x)\exp\{\theta^{\mathrm{T}}\phi(x)\}\phi(x)\mathrm{d}x}{\exp(A(\theta))} \\
&= \frac{\int h(x)\exp\{\theta^{\mathrm{T}}\phi(x)\}\phi(x)\mathrm{d}x}{\exp(A(\theta))} \\
&= \int h(x)\exp\{\theta^{\mathrm{T}}\phi(x) - A(\theta)\}\phi(x)\mathrm{d}x \\
&= \int p(x|\theta)\phi(x)\mathrm{d}x \\
&= E_\theta[\phi(x)]
\end{aligned}$$

第二个性质同样与对数分割函数有关：$A(\theta)$ 的二阶导等于 $\phi(x)$ 的方差。这听上去就更不可思议了，但是它确实成立，可以得到

$$\begin{aligned}
\frac{\partial^2 A(\theta)}{\partial \theta \partial \theta^{\mathrm{T}}} &= \int h(x)\exp(\theta^{\mathrm{T}}\phi(x) - A(\theta))\phi(x)(\phi(x) - \frac{\partial A(\theta)}{\partial \theta})\mathrm{d}x \\
&= \int p(x|\theta)\phi(x)(\phi(x) - E_\theta[\phi(x)])\mathrm{d}x \\
&= \int p(x|\theta)\phi^2(x)\mathrm{d}x - E_\theta[\phi(x)]\int p(x|\theta)\phi(x)\mathrm{d}x \\
&= E_\theta[\phi^2(x)] - E_\theta^2[\phi(x)] \\
&= \mathrm{Var}_\theta[\phi(x)]
\end{aligned}$$

为了证明这个性质的效果，我们以伯努利分布为例进行求解，已知 $A(\phi) = \log(1 + \exp(\theta))$，$\phi(x) = x$，可以得到

$$E[\phi(x)] = \frac{\partial A(\phi)}{\partial \phi} = \frac{\mathrm{e}^\theta}{1 + \mathrm{e}^\theta} = \frac{1}{1 + \mathrm{e}^{-\theta}} = p$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$
\begin{aligned}
\mathrm{Var}[\phi(x)] &= \frac{\partial^2 A(\theta)}{\partial\theta\partial\theta^{\mathrm{T}}} \\
&= \frac{\partial}{\partial\theta}\left(\frac{1}{1+\mathrm{e}^{-\theta}}\right) \\
&= -(1+\mathrm{e}^{-\theta})^{-2}\times\mathrm{e}^{-\theta}(-1) \\
&= \frac{1}{1+\mathrm{e}^{-\theta}}\left(1-\frac{1}{1+\mathrm{e}^{-\theta}}\right) \\
&= p(1-p)
\end{aligned}
$$

我们发现利用上面的性质得到了正确的结果，这两个性质告诉我们，将概率分布以指数家族的形式表达后，$\phi(x)$ 的期望与方差实际上拥有两种解法，一种是直接使用这个函数进行求解，另一种则是采用对数分割函数进行求解。在真实的问题中，我们可以根据实际情况选择一种更简单的方法进行求解。

在接下来的章节里，我们要使用指数家族的另一种简单的形式，即：

$$
p(x|\theta) = \frac{1}{Z(\theta)}h(x)\mathrm{e}^{\theta^{\mathrm{T}}\phi(x)}
$$

可以看出这里的 $Z(\theta)$ 就是由 $A(\theta)$ 转变得到的。感兴趣的读者可以研究这两个公式的关系。

15.1.2 最大熵模型的推导

介绍完指数家族的基本概念，让我们回到最大熵模型。2.4 节介绍过熵的概念，它表示随机变量的不确定性。换句话说，它表示了知道随机变量结果的“代价”。熵的值越大，代表随机变量越“捉摸不定”，反之则越确定。人类在面对未知事物时，往往也倾向于产生怀疑的态度。

以最经典的投掷骰子的问题为例。当我们以“正常的手法”投掷出一枚骰子时，会认为六个面朝上的概率是相等的，因为没有任何证据表明其中的某一个面将会以更高的概率出现。这种平等看待所有未知情况的思想其实暗含了最大熵的思想，因为使熵最大的方法就是让所有的情况具有相等的概率，这样也最难推断出结果。如果我们知道其中的某一面会以 1/3 的概率出现，那么剩下的五个面将共享 2/3 的概率。此时，由于我们没有其他关于另外五面的信息，因此按照最大熵的思想，我们只能认为剩下的五个面将以等概率出现。

对机器学习来说，我们同样需要这种思想来帮助我们确定模型的形式。在我们建立一个概率模型时，往往会伴随一些约束，例如对数值的直接约束，要求与训练数据尽

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

可能地接近等。有时，这些约束的存在并不能得到一个唯一的模型，满足这些条件的模型有很多种，这些模型在约束上的表现是基本一致的，而在没有约束的子空间上则表现得不尽相同。那么，根据最大熵的思想，没有约束的子空间应该拥有均等的概率。这样我们就可以确定一个唯一的概率分布，而这个概率分布就是我们想要的。

介绍完最大熵的思想，下面介绍模型的具体构建形式。假设一组数据 (x, y)，它的输出是 y，y 属于集合 $\boldsymbol{Y}$；它的输入是 x，x 来自集合 $\boldsymbol{X}$。我们的目标是构建一个模型，来表示 x 和 y 之间的关系，这里采用条件概率分布 $p(y|x)$ 进行构建。假设数据是离散的，最简单直接的方法就是基于这些数据，利用直方图计数的方式构建一个分布。对应的公式为

$$\tilde{p}(x, y) = \frac{1}{N}\text{count}(x, y)$$

其中 N 表示数据总量，count() 表示给定的数据样本出现了多少次。这其实就是最简单的计算频率的公式。如果数据量十分充足，根据大数定理，直接用这个运算结果作为模型的参数就可以了，它可以很好地表达数据的真实分布。

在实际中，由于问题的规模一般很大，数据总是不够，我们仍然需要用建模的方法达到更好的模型效果。所谓更好的模型效果指的是机器学习中经常提到的泛化性，我们希望在数据缺失的区域也能达到很好的效果。由于数据可能来自高维的空间，为了将数据空间进行压缩，我们要通过**特征**的形式将数据进行转换。特征的一般形式为

$$f(x, y) = \begin{cases} 1 & \text{如果 } x = x_0 \text{ 且 } y = y_0 \\ 0 & \text{否则} \end{cases}$$

我们可以根据问题的具体形式设立不同的特征，为每个特征定义不同的满足条件，这样每一对原始数据就可以通过这些特征转换成有限维空间中的数据。特征在训练数据上也具有一定的统计特性，我们定义“特征 $f(x, y)$ 关于训练样本分布 $\tilde{p}(x, y)$ 的期望”为

$$\tilde{p}(f) = \sum_{x,y} \tilde{p}(x, y) f(x, y) = \sum_{x,y} \frac{1}{N}\text{count}(x, y) f(x, y)$$

模型要建立概率分布 $p(y|x)$ 与训练样本分布的特征期望对应，我们可以得到特征关于模型分布的期望值：

$$p(f) = \sum_{x,y} \tilde{p}(x) p(y|x) f(x, y)$$

我们希望模型能学习到数据中的分布特性，于是我们可以设立一个约束：让模型分布的特征期望和训练样本分布的特征期望相等，于是我们就有了约束：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$\tilde{p}(f) = p(f)$$

公式展开后可以得到

$$\sum_{x,y} \tilde{p}(x,y) f(x,y) = \sum_{x,y} \tilde{p}(x) p(y|x) f(x,y)$$

除了保持这个约束，我们还希望模型的概率分布能够满足最大熵的约束，于是我们的目标函数就是尽可能地增加概率分布的熵，于是可以得到公式

$$H(p) = -\sum_{x,y} \tilde{p}(x) p(y|x) \log p(y|x)$$

把所有的约束和目标完整地列出来，就得到了最大熵模型的完整形式：

$$\begin{aligned} p^* &= \operatorname{argmax}_p H(p) = \operatorname{argmax}_p -\sum_{x,y} \tilde{p}(x) p(y|x) \log p(y|x) \\ \text{s.t.} \quad & p(y|x) \geqslant 0, \forall x, y \\ & \sum_y p(y|x) = 1, \forall x \\ & \sum_{x,y} \tilde{p}(x) p(y|x) f(x,y) = \sum_{x,y} \tilde{p}(x,y) f(x,y), i \in \{1, 2, \ldots, n\} \end{aligned}$$

完成了问题的定义，下面就来解决这个问题。我们采用朗格朗日乘子法，为每一个特征引入一个参数 λ_i，得到拉格朗日函数 $\xi(p, \Lambda, \gamma)$：

$$\begin{aligned} \xi(p, \Lambda, \gamma) = & -\sum_{x,y} \tilde{p}(x) p(y|x) \log p(y|x) + \sum_{i=1}^{n} \lambda_i \Big(\sum_{x,y} \tilde{p}(x) p(y|x) f_i(x,y) \\ & - \sum_{x,y} \tilde{p}(x,y) f_i(x,y) \Big) + \gamma \Big(\sum_y p(y|x) - 1 \Big) \end{aligned}$$

其中 Λ 表示乘子 $\lambda_1, \ldots, \lambda_n$ 的集合。我们假设 Λ 和 γ 不变，对 p 求导，可以得到

$$\frac{\partial \xi}{\partial p(y|x)} = -\tilde{p}(x)(\log p(y|x) + 1) + \sum_{i=1}^{n} \lambda_i \tilde{p}(x) f_i(x,y) + \gamma$$

令导数为 0，可以得到关于 p 的极值：

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$-\tilde{p}(x)(\log p(y|x)+1)+\sum_{i=1}^{n}\lambda_i\tilde{p}(x)f_i(x,y)+\gamma=0$$

对公式进行整理，可以得到

$$p(y|x)=\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))\exp(\frac{\gamma}{\tilde{p}(x)}-1)$$

如果我们再次使用前面关于概率总和为 1 的约束：

$$\sum_y p(y|x)=1,\forall x$$

将公式进行带入求解，就可以得到

$$\begin{aligned}\sum_y p(y|x)&=\sum_y\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))\exp(\frac{\gamma}{\tilde{p}(x)}-1)\\ \rightarrow\quad 1&=\sum_y\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))\exp(\frac{\gamma}{\tilde{p}(x)}-1)\\ \rightarrow\exp(\frac{\gamma}{\tilde{p}(x)}-1)&=\frac{1}{\sum_y\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))}\end{aligned}$$

将这个公式带入上面的公式，得到

$$p(y|x)=\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))\frac{1}{\sum_y\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))}$$

令 $Z(x)$ 为

$$Z(x)=\sum_y\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))$$

模型的最优解为

$$p(y|x)=\frac{1}{Z(x)}\exp(\sum_{i=1}^{n}\lambda_i f_i(x,y))$$

看到最大熵模型的最优解形式时，相信读者会有种似曾相识的感觉，这不就是我们前面见到的指数家族的结构形式吗？经过前面拉格朗日乘子法的计算，变量由模型 p 变成了乘子 λ，而此时这个乘子更像是每一个特征的权重项，我们通过为每个特征乘以一个权重判断数据所属的 y 的概率值。

书籍是人类进步的阶梯

既然我们已经得到了概率分布的形式，下面就使用最大似然法求解模型中的参数。将模型的结构展开，可以得到

$$p(y|x,\lambda)=\frac{\exp[\sum_i \lambda_i f_i(x,y)]}{\sum_{y'}\exp[\sum_i \lambda_i f_i(x,y')]}$$

我们的目标是最大化对数似然 $\log p(y|x,\lambda)$，可以得到

$$\begin{aligned}\log p(Y|X)&=\sum_{(x,y)\in(X,Y)}\log p(y|x,\lambda)\\&=\sum_{(x,y)\in(X,Y)}\log\frac{\exp[\sum_i \lambda_i f_i(x,y)]}{\sum_{y'}\exp[\sum_i \lambda_i f_i(x,y')]}\\&=\sum_{(x,y)\in(X,Y)}\sum_i \lambda_i f_i(x,y)-\sum_{(x,y)\in(X,Y)}\log\sum_{y'}\exp[\sum_i \lambda_i f_i(x,y')]\end{aligned}$$

对公式进行求导，可以得到

$$\begin{aligned}&\frac{\partial \log P(Y|X)}{\partial \lambda_i}\\&=\sum_{(x,y)\in(X,Y)} f_i(x,y)-\sum_{(x,y)\in(X,Y)}\frac{1}{\sum_{y''}\exp[\sum_i \lambda_i f_i(x,y'')]}\frac{\partial\sum_{y'}\exp[\sum_i \lambda_i f_i(x,y')]}{\partial\lambda_i}\\&=\sum_{(x,y)\in(X,Y)} f_i(x,y)-\sum_{(x,y)\in(X,Y)}\frac{1}{\sum_{y''}\exp[\sum_i \lambda_i f_i(x,y'')]}\sum_{y'}\exp[\sum_i \lambda_i f_i(x,y')]f_i(x,y')\\&=\sum_{(x,y)\in(X,Y)} f_i(x,y)-\sum_{(x,y)\in(X,Y)}\sum_{y'}\frac{\exp[\sum_i \lambda_i f_i(x,y')]}{\sum_{y''}\exp[\sum_i \lambda_i f_i(x,y'')]}f_i(x,y')\\&=\sum_{(x,y)\in(X,Y)} f_i(x,y)-\sum_{(x,y)\in(X,Y)}\sum_{y'}p(y'|x,\lambda)f_i(x,y')\end{aligned}$$

可以看出，最大似然的梯度等于训练数据分布的特征期望与模型的特征期望的差。当梯度为 0 时，得到的概率分布刚好满足约束条件：

$$\sum_{(x,y)\in(X,Y)} f_i(x,y)=\sum_{(x,y)\in(X,Y)}\sum_{y'}p(y'|x,\lambda)f_i(x,y')$$

所以我们可以采用梯度下降的方法不断迭代更新参数 λ。

最后让我们验证前面留下的一个疑问，采用这种计算方法是否能保证求解结果唯一呢？下面我们来证明。我们已经通过优化得到了一个概率模型 p，假设存在另外一个指数家族的概率模型 $\hat{p}(y|x)=\exp(\hat{\lambda}^{\mathrm{T}}f(x,y)-A(\lambda))$（我们将 Z 转换成了 A），也能够

书籍是人类进步的阶梯

在满足约束的条件下最大化目标函数，且参数与 p 不同，那么可以得到

$$\begin{aligned}
H(p(x,y)) &= -\sum_x p(x)\sum_y p(y|x)\log p(y|x) \\
&= -\sum_x p(x)\sum_y p(y|x)\log(\frac{p(y|x)}{\hat{p}(y|x)} \times p(\hat{y}|x)) \\
&= -\sum_x p(x)[\sum_y p(y|x)\log\frac{p(y|x)}{\hat{p}(y|x)} - \sum_y p(y|x)\log\hat{p}(y|x)] \\
&= -\sum_x p(x)[\mathrm{KL}(p(y|x)\|\hat{p}(y|x)) - \sum_y p(y|x)[\hat{\lambda}^{\mathrm{T}} f(x,y) - A(\lambda)]]
\end{aligned}$$

因为两个模型都达到了最优，所以可以得到两个模型的特征期望与训练数据相同：

$$\sum_x p(x)\sum_y p(y|x)f(x,y) = \sum_x\sum_y \hat{p}(y|x)f(x,y) = \sum_x f(x,y)$$

所以可以得到

$$\begin{aligned}
&= -\mathrm{KL}(p(x,y)\|\hat{p}(x,y)) - \sum_x p(x)[\sum_y p(y|x)\hat{\lambda}^{\mathrm{T}} f(x,y) - \sum_y p(y|x)A(\theta)] \\
&= -\mathrm{KL}(p(x,y)\|\hat{p}(x,y)) - \sum_x p(x)[\sum_y \hat{p}(y|x)\hat{\lambda}^{\mathrm{T}} f(x,y) - \sum_y \hat{p}(y|x)A(\theta)] \\
&= -\mathrm{KL}(p(x,y)\|\hat{p}(x,y)) - \sum_x p(x)\sum_y \hat{p}(y|x)\log[\exp[\hat{\lambda}^{\mathrm{T}} f(x,y) - A(\theta)]] \\
&= -\mathrm{KL}(p(x,y)\|p(\hat{x},y)) - \sum_x p(x)\sum_y \hat{p}(y|x)\log\hat{p}(y|x) \\
&= -\mathrm{KL}(p(x,y)\|\hat{p}(x,y)) + H(\hat{p}(x,y))
\end{aligned}$$

因为 $\mathrm{KL}(p(x,y)\|\hat{p}(x,y)) \geqslant 0$，所以 $H(p) \geqslant H(\hat{p})$，且当 $p = \hat{p}$ 时，两者的熵相等，这样就可以证明最优解是唯一的。

15.1.3 最大熵模型的实现

经过前面的推导，我们已经求出了最大熵模型的目标梯度。接下来我们将给出它的实现算法。最大熵模型的梯度下降法实现起来相对简单。令 X 的数量为 $\dim_x$，Y 的数量为 $\dim_y$，于是我们可以定义 $\dim_x \times \dim_y$ 个特征，每一个特征可以指示一组数据是否存在：

书籍是人类进步的阶梯

$$f_{i,j}(x,y)=\begin{cases}1 & i=x \text{ 且 } j=y\\ 0 & \text{其他情况}\end{cases}$$

模型拥有 $\dim_x \times \dim_y$ 个参数 $\lambda_{i,j}$，每一个参数表示对应特征的权重。如果一个样本拥有 N 个特征，那么我们可以得到某个样本 x 关于结果 y_j 的概率：

$$p(y_j|x)=\frac{\exp(\sum_i \lambda_{i,j} f_{i,j}(x_i,y_j))}{\sum_j \exp(\sum_i \lambda_{i,j} f_{i,j}(x_i,y_j))}$$

知道了这个概率，就能计算出一个样本对所有特征的积累量，其中每一个特征积累量为

$$f_{i,j}(x)=\sum_j p(y_j|x) f_{i,j}(x_i,y_j)$$

那么，也就可以求出一批样本对所有特征的积累量，其中每一个特征积累量为

$$f_{i,j}(X)=\sum_{(x,y)\sim(X,Y)} \sum_j p(y_j|x) f_{i,j}(x_i,y_j)$$

求解出所有的特征累积量，就可以轻松求出模型的梯度，完整的算法如下所示。

算法 最大熵似然法

Input

模型参数 $\lambda_{i,j}$，特征 $\boldsymbol{f}_{i,j}$

Start

for $i=1,\ldots,N$ **do**

初始化训练样本特征期望矩阵 $\tilde{\boldsymbol{f}}$ 和模型特征期望矩阵 $\boldsymbol{f}$，二者的维度为 $|F|\times|Y|$。

取出一个批次的数据，根据公式计算特征期望，其中 i 和 j 分别表示 x 和 y 的下标：

$\tilde{\boldsymbol{f}}_{i,j}=\sum_{(x,y)\sim(X,Y)} \tilde{f}_{i,j}(x,y)$

求出每一个特征在模型的积累量：

$f_{i,j}(X)=\sum_{(x,y)\sim(X,Y)} \sum_j p_j(y_j|x) f_{i,j}(x_i,y_j)$

其中 $p_j(y_j|x)=\frac{\exp(\sum_i \lambda_{i,j} f_{i,j}(x_i,y_j))}{\sum_j \exp(\sum_i \lambda_{i,j} f_{i,j}(x_i,y_j))}$

计算每一个参数的更新量并对参数进行更新：

$\Delta\lambda_{i,j}=\alpha(\tilde{\boldsymbol{f}}_{i,j}-\boldsymbol{f}_{i,j})$

end for

15.2 最大熵反向强化学习

经过了 15.1 节的铺垫，本节介绍最大熵模型在反向强化学习中的应用，算法来自论文 *Maximum entropy inverse reinforcement learning*[1]。再次回顾反向强化学习的问题定义：对于某个强化学习的任务，我们观察到专家的状态–行动轨迹：$\zeta = \{\boldsymbol{s}_i, \boldsymbol{a}_i\}_N$，但我们并不知道任务中回报函数的形式。我们希望使用这些轨迹信息学习到回报函数 $r(\boldsymbol{f}_{\boldsymbol{s}_j}; \theta)$，其中 $\boldsymbol{f}_{\boldsymbol{s}_j}$ 是由状态转换得到的特征，θ 是回报函数的参数。如果回报函数是一个线性函数，那么回报函数可以写作

$$r(\boldsymbol{f}_{\boldsymbol{s}_j}) = \theta^{\mathrm{T}} \boldsymbol{f}_{\boldsymbol{s}_j} = \sum_i \theta_i \boldsymbol{f}_i$$

那么轨迹的长期回报就可以写作

$$r(\boldsymbol{f}_\zeta) = \sum_{\boldsymbol{s}_j \in \zeta} \theta^{\mathrm{T}} \boldsymbol{f}_{\boldsymbol{s}_j} = \theta^{\mathrm{T}} \sum_{\boldsymbol{s}_j \in \zeta} \boldsymbol{f}_{\boldsymbol{s}_j} = \theta^{\mathrm{T}} \boldsymbol{f}_\zeta$$

这样就可以把 $\boldsymbol{f}_\zeta$ 称为特征积累量，每一条轨迹被称为 $\tilde{\zeta}_i$，我们假设所有的轨迹起始于同一个状态，并将所有的轨迹聚合起来，就可以用这些轨迹得到价值期望的估计：

$$v(\boldsymbol{f}) = E_{\tilde{\zeta}_i}[r(f_{\tilde{\zeta}_i})] \simeq \frac{1}{m} \sum_i \theta^{\mathrm{T}} \boldsymbol{f}_{\tilde{\zeta}_i} = \theta^{\mathrm{T}} \frac{1}{m} \sum_i f_{\tilde{\zeta}_i} = \theta^{\mathrm{T}} \tilde{\boldsymbol{f}}$$

所以 $\tilde{\boldsymbol{f}}$ 也可以表示特征积累的期望。

定义策略模型为 $P_\pi(\zeta_i)$，同最大熵模型中的约束条件类似，我们希望交互得到的轨迹的长期回报等同于专家的轨迹的长期回报，于是可以得到

$$E_{\zeta_i \sim \pi}[r(f_\zeta)] = \theta^{\mathrm{T}} \sum_i P_\pi(\zeta_i) \boldsymbol{f}_{\zeta_i} = \theta^{\mathrm{T}} \tilde{\boldsymbol{f}}$$

由于等式两边的 θ 相同，可以得到

$$\sum_i P_\pi(\zeta_i) \boldsymbol{f}_{\zeta_i} = \frac{1}{m} \sum_i f_{\tilde{\zeta}_i}$$

这样我们就得到了一个和最大熵模型类似的约束：策略的轨迹特征积累量的期望应该与专家的相等。这个条件并不是一个很强的约束，满足这个条件的策略模型可能有很多，而这些策略在完整的问题空间中可能各有优劣，所以我们仍然有可能学习出

一个较差的模型。为了解决这个问题，我们需要再添加最大熵的约束。如果有两个策略 π_1 和 π_2，它们的特征累积量期望是相同的，但它们的参数不同，那么至少存在一条轨迹，使得两个策略对它的“喜好”不同，一个策略有较高的概率产生这条轨迹，而另一个策略产生这条轨迹的概率较低。如果我们可以将轨迹对某个策略的概率用形式化的方法表示，就可以将目标函数定义为最大化概率分布的熵，从而区别这些策略。

如果 MDP 是一个确定的 MDP，也就是说当策略选定了行动后，环境会转换到确定的状态，那么轨迹的长期回报只和策略对轨迹的概率有关。于是我们可以使用轨迹的长期回报表示轨迹出现的概率，这个表示是非常合理的，如果轨迹获得的长期回报比较高，那么策略应该以较高的概率出现这条轨迹，这样也更符合最大化长期回报的目标。于是轨迹的概率可以表示为

$$P(\zeta_i|\theta)=\frac{1}{Z(\theta)}\mathrm{e}^{\theta^{\mathrm{T}}\boldsymbol{f}_{\zeta_i}}=\frac{1}{Z(\theta)}\mathrm{e}^{\theta^{\mathrm{T}}\sum_{s_j\in\zeta_i}\boldsymbol{f}_{s_j}}$$

其中 $Z(\theta)=\sum_i \mathrm{e}^{\theta^{\mathrm{T}}\boldsymbol{f}_{\zeta_i}}$，得到了轨迹的概率形式，我们就可以构建基于最大熵的目标函数了。确定的 MDP 状态转换图如图 15-1 所示。

当然，在实际问题中，我们也会遇到非确定的 MDP 问题。也就是说，当策略确定了行动后，下一个状态并不确定，仍然需要环境的状态转移概率确定，这样我们就必须把这部分随机性考虑进去。轨迹的概率由策略和状态转换概率共同决定。那么，我们使用某个策略与环境交互，在环境随机性的影响下，一条轨迹可能是多个行动序列的结果。这就是非确定的 MDP 问题的特点，如图 15-2 所示。

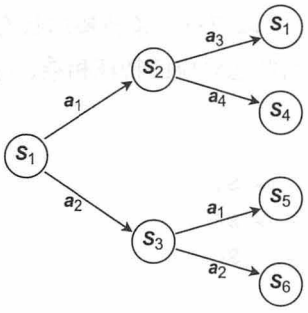


图 15-1 确定的 MDP 状态转换

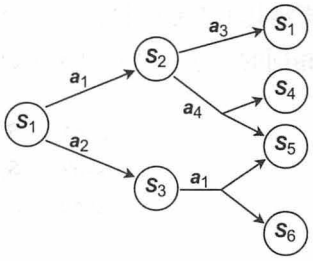


图 15-2 非确定的 MDP 状态转换

要想表示一条轨迹的长期回报期望，首先要找到轨迹对应的所有行动序列。令一条行动序列为 $\boldsymbol{o}$，所有行动序列的集合为 $\boldsymbol{O}$，就可以使用指示变量 $I_{\zeta\in\boldsymbol{o}}$ 表示轨迹是否可以通过某一条行动序列生成。当这个值等于 1 时，表示轨迹可以通过行动序列生成，反之则无法生成。同时，我们用 $P_T(\zeta|\boldsymbol{o})$ 表示当行动序列确定时，状态最终会转换成序列 ζ 的概率。于是轨迹的概率可以表示为

$$P(\zeta|\theta,T)=\sum_{o\in O}P_T(\zeta|o)\frac{e^{\theta^T f_\zeta}}{Z(\theta,o)}I_{\zeta\in o}$$

其中 $Z(\theta,o)$ 的定义也随之改变。上面的公式虽然是正确的，但是由于有指示变量的存在，这个公式很难计算。我们需要将上面的公式近似为一个可解的形式：

$$P(\zeta|\theta,T)\simeq\frac{e^{\theta^T f_\zeta}}{Z(\theta,o)}\prod_{s_{t+1},a_t,s_t\in\zeta}P_T(s_{t+1}|a_t,s_t)$$

经过上面的推导，得到了模型对轨迹的概率，于是我们可以采用最大似然法进行优化，假设问题是确定的 MDP，对应的目标函数为

$$\theta^*=\text{argmax}_\theta L(\theta)=\text{argmax}\sum_\zeta \log p(\zeta|\theta,T)$$

由于公式的形式和最大熵模型十分一致，于是我们省略了推导过程，直接给出参数梯度的计算公式：

$$\nabla L(\theta)=\tilde{f}-\sum_\zeta p(\zeta|\theta,T)f_\zeta$$

这样的形式实际上仍不容易计算，$p(\zeta|\theta,T)$ 和 f_ζ 这两项都显得有些复杂。下面我们来化简其中一个项目，并将复杂的计算“转移”到另一个项目。我们要考虑的问题是确定 MDP 的问题，即行动可以直接决定下一时刻的状态。这样当初始状态、策略和回报函数确定时，未来的轨迹也就确定下来。所以我们可以把每一条轨迹的状态分别列出来，再将其中重复的状态合并，将这些状态的概率与状态对应的特征相乘，得到的结果也是相同的。这一过程的变换如图 15-3 所示。

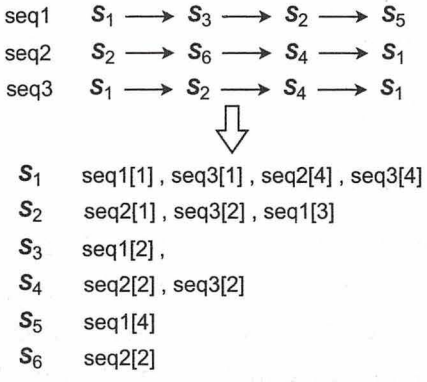


图 15-3　期望状态访问频率转换

图 15-3 中包含 3 个序列，每个序列都包含 4 个状态。我们以状态 S_1 为例，它出现在 4 个位置上，在计算期望时，我们实际上要让这 4 个位置的状态概率分别乘以状态对应的特征。由于这 4 个位置的特征值相同，我们可以将它们合并为同一项。只要能计算出 4 个位置的状态概率和，再将其乘以特征值，就可以求出序列中关于这个状态的特征期望，其他的状态也可以依次得到。

这样我们就得到了一个新的计算公式

$$\nabla L(\theta) = \tilde{\boldsymbol{f}} - \sum_{\boldsymbol{s}_i} D_{\boldsymbol{s}_i} \boldsymbol{f}_{\boldsymbol{s}_i}$$

其中 $D_{\boldsymbol{s}_i}$ 被称为状态访问频率期望（Expected State Visitation Frequency），我们将计算的重点放在这一项。从任意一个状态出发，根据策略的频率和确定的状态转换概率推演，可以得到任意一个时刻状态出现的概率，之后将每一个时刻状态出现的概率加起来，就可以得到 $D_{\boldsymbol{s}_i}$。

从上面的分析来看，目前我们已经知道了状态转换概率，唯一不清楚的就是策略，于是我们先求解策略。最大熵反向强化学习的论文 [1] 中介绍了一种基于前向后向计算的方法，它可以先通过反向计算求出策略模型，再进行前向计算得到状态访问频率期望。

首先设定在最后一刻 T 所有状态的出现值为 1，就可以计算 $T-1$ 时刻某个状态下执行某个行动的概率 $P_{T-1}(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i)$，可以得到

$$\begin{aligned} P_{T-1}(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i) &= \frac{P(\boldsymbol{s}_i, \boldsymbol{a}_{i,j})}{P(\boldsymbol{s}_i)} \\ &= \frac{\sum_k P(\boldsymbol{s}_i, \boldsymbol{a}_{i,j}, \boldsymbol{s}_k)}{\sum_{\boldsymbol{a}_{i,j}} \sum_k P(\boldsymbol{s}_i, \boldsymbol{a}_{i,j}, \boldsymbol{s}_k)} \\ &= \frac{\sum_k P(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i) p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j})}{\sum_{\boldsymbol{a}_{i,j}} \sum_k P(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i) p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j})} \end{aligned}$$

前面我们提到轨迹出现的概率和回报成正相关，于是我们可以使用回报替代概率值，这样就可以得到

$$= \frac{\sum_k \mathrm{e}^{\mathrm{reward}(\boldsymbol{s}_i|\theta)} p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j})}{\sum_{\boldsymbol{a}_{i,j}} \sum_k \mathrm{e}^{\mathrm{reward}(\boldsymbol{s}_i)|\theta} p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j})}$$

如果令 $Z_{\boldsymbol{a}_{i,j}} = \sum_k \mathrm{e}^{\mathrm{reward}(\boldsymbol{s}_i|\theta)} p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j})$，公式就可以变为

$$= \frac{Z_{\boldsymbol{a}_{i,j}}}{\sum_{\boldsymbol{a}_{i,j}} Z_{\boldsymbol{a}_{i,j}}}$$

从公式中可以发现，分子和分母存在一定的重复，这部分计算可以共享。当时刻转换到 $T-2$ 时，我们也可以得到类似的结果：

$$P_{T-2}(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i) = \frac{\sum_k \mathrm{e}^{\mathrm{reward}(\boldsymbol{s}_i|\theta)} p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j}) Z_{\boldsymbol{s}_k}}{\sum_{\boldsymbol{a}_{i,j}} \sum_k \mathrm{e}^{\mathrm{reward}(\boldsymbol{s}_i)|\theta} p(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j}) Z_{\boldsymbol{s}_k}}$$

这样就得到了一个迭代的公式形式，随着迭代的轮数不断增加，策略估计值会变得越来越稳定，越来越接近真实的策略，这样我们就可以完成策略的计算，计算过程如图 15-4 所示。

通过反向计算得到策略值后，再进行前向计算，就可以得到每一个时刻状态的访问频率，这部分的计算过程如图 15-5 所示。

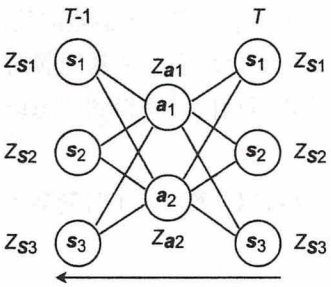


图 15-4　反向计算过程

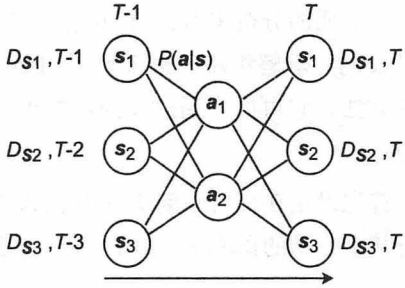


图 15-5　前向计算过程

这样就可以得到状态访问频率期望 $D_{\boldsymbol{s}_i}$，使接下来梯度的计算变得容易。前向后向算法如下所示。

算法　状态访问频率期望计算

反向计算：

（1）设置 $Z_{\boldsymbol{s}_i} = 1$

（2）重复计算 N 个时间迭代过程

$Z_{\boldsymbol{a}_{i,j}} = \sum_k P(\boldsymbol{s}_k|\boldsymbol{s}_i, \boldsymbol{a}_{i,j}) \mathrm{e}^{\mathrm{reward}(\boldsymbol{s}_i|\theta)} Z_{\boldsymbol{s}_k}$

$Z_{\boldsymbol{s}_i} = \sum_{\boldsymbol{a}_{i,j}} Z_{\boldsymbol{a}_{i,j}}$

（3）概率计算

$P(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i) = \frac{Z_{\boldsymbol{a}_{i,j}}}{Z_{\boldsymbol{s}_i}}$

（4）前向计算

设置 $D_{\boldsymbol{s}_i,t} = P(\boldsymbol{s}_i = \boldsymbol{s}_{\text{initial}})$

for $t = 1$ to N;

$D_{\boldsymbol{s}_i,t+1} = \sum_{\boldsymbol{a}_{i,j}} \sum_k D_{\boldsymbol{s}_k,t} P(\boldsymbol{a}_{i,j}|\boldsymbol{s}_i) P(\boldsymbol{s}_k|\boldsymbol{a}_{i,j}, \boldsymbol{s}_i)$

（5）加和频率

$D_{\boldsymbol{s}_i} = \sum_t D_{\boldsymbol{s}_i,t}$

了解了最大熵算法的核心部分，就可以得到完整的算法。剩下的部分和最大熵似然法类似，这里不再展开。论文 *Maximum Entropy Deep Inverse Reinforcement Learning*[2] 中还介绍了深度最大熵反向强化学习算法，整体思想与最大熵算法比较相近，主要的改进是将其中的模型换成了深层模型，感兴趣的读者可以自行阅读。

15.3 GAIL

本节介绍另一个反向强化学习的方法，它的名称叫作 GAIL（Generative Adversarial Imitation Learning），来自同名论文 [3]。从名字上可以看出，这个方法使用 GAN（Generative Adversarial Network）完成 Imitation Learning 的工作。近一两年，与 GAN 相关的研究发展十分迅速，也催生了许多应用方面的扩展，GAIL 就是其中之一。本节先介绍 GAN 的基础知识，然后介绍 GAN 与反向强化学习结合的方法。

15.3.1 GAN 的基本概念

GAN 是一个实现生成模型的网络结构，它的结构比较复杂，其中包含了一对子模型：生成模型和判别模型。生成模型用于生成目标概率分布中的样本，而判别模型用于判断样本是否来自真实数据分布。GAN 最早被应用于生成图像，由于图像数据本身比较复杂，它的分布也比较复杂，因此很多常规的生成模型很难生成令人满意的图像。而 GAN 通过隐式概率的形式生成图像，同时配合判别模型，可以生成更高质量的图像。

我们用字母 D 代表判别模型，它的输入是数据空间内的任意一张图像 x，输出是一个概率值，表示这张图像属于真实数据的概率。字母 G 代表生成模型，它的输入是一个随机变量 z，z 服从某种分布，输出是一张图像 $G(z)$，如果它生成的图像经过模型 D 后的概率值很高，就说明生成模型已经比较好地掌握了数据的分布模式，可以产生符合要求的样本；反之则没有达到要求，还需要继续训练。

两个模型的目标公式如下所示。

仅供非商业用途或交流学习使用

（1）判别模型的目标是最大化公式 $E_x[D(x)]+E_z[1-D(G(z))]$，也就是甄别出哪些图来自真实数据分布，哪些是由生成模型生成的。

（2）生成模型的目标是最大化公式 $E_z[D(G(z))]$，也就是让自己生成的图被判别模型判断为来自真实数据分布。

从目标公式可以看出两个模型之间的联系，如果生成模型生成的图像和真实的图像有区别，判别模型要给前者判定比较低的概率。这里可以举个形象的例子，假设 x 是一种商品，D 是商品的检验方，负责检验商品是否是正品；G 是一家山寨公司，希望根据拿到手的一批产品 x 研究出生产山寨商品 x 的方式。对 D 来说，不管 G 生产出来的产品多像正品，都应该被判定为赝品。G 的技术水平不高，生产出来的产品漏洞百出，只有不断地提高技术水平，才有可能迷惑检验方。

基于上面的例子，两个模型的目标可以统一成一个充满硝烟味的目标函数。

$$\min_G \max_D V(D,G)=E_x[\log D(x)]+E_z[\log(1-D(G(z)))]$$

上面这个公式对应的模型架构如图 15-6 所示。

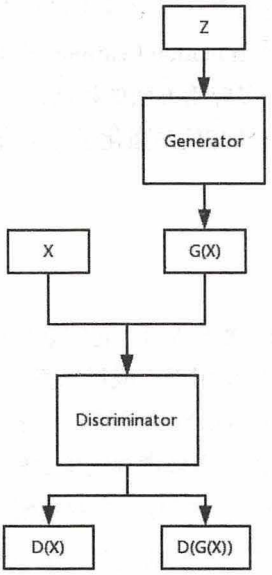


图 15-6　GAN 的基本形式

对应的模型学习算法的伪代码如下所示：

```
def GAN(G,D,X):
    # G表示生成模型
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
# D表示判别模型
# X表示训练数据
for iter in range(MAX_ITER):
    for step in range(K):
        x = data_sample(X)
        z = noise_sample()
        optimize_D(G, D, x, z)
    z = noise_sample()
    optimize_G(G, D, z)
```

上面的代码从宏观层面介绍了模型的优化方法，其中 K 表示判别模型 D 的迭代次数，K 一般大于等于 1。从上面的公式可以看出，两个模型的目标是对立的。生成模型希望最大化自己生成的图像的似然，判别模型希望最大化原始数据的似然的同时，能够最小化 G 生成的图像的似然。既然是对立的，两个模型经过训练产生的能力就可能有很多种情况。它们既可能上演“魔高一尺，道高一尺”的竞争戏码，在竞争中共同成长，最终产生两个强大的模型；也可能产生一个强大的模型，将另一方完全压倒。

（1）如果判别模型太过强大，那么生成模型会有两种情况：一种情况是发现自己完全被针对，模型参数无法优化；另一种情况是发现判别模型的一些漏洞后，它的模型将退化，不管输入是什么，输出统一变成之前突破了判别模型防线的那几类结果。这种情况被称为 Mode Collapse，有点像一个复杂强大的模型崩塌成一个简单弱小的模型，这样的模型即使优化结果很好，也不能拿去使用。

（2）如果判别模型不够强大，无法精确判别一个样本是否来自真实数据，而生成模型又是按照它的判别结果进行调节，那么生产出的产品质量就不太有保证，这同样不是我们想看到的结果。

总而言之，对抗是 GAN 这个模型要面对的一个大问题。后续基于 GAN 的很多改进模型也都在试图解决这个问题，每一个模型有自己的改进策略，这些改进方法的详细信息请读者自行查阅了解。

15.3.2 GAN 的训练分析

我们从目标函数出发对 GAN 的训练过程进行分析。我们先证明第一步：当生成模型固定时，判别模型的最优形式。首先将目标函数做变换：

$$\max_D V(D,G) = E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

$$= \int_x p_r(x)\log D(x)\mathrm{d}x + \int_z p_g(z)\log(1 - D(G(z)))\mathrm{d}z$$

由于组成公式的两部分积分的区域不同，会对后面的计算造成困难，所以先将两个积分区域统一。我们将生成图像 $G(z)$ 的分布与真实图像 x 的分布做一个投射，只要判别式能够在真实数据出现的区域保证判别正确即可，于是公式就变为

$$\begin{aligned} &= \int_x p_r(x)\log D(x)\mathrm{d}x + \int_x p_g(x)\log(1 - D(x))\mathrm{d}x \\ &= \int_x [p_r(x)\log D(x) + p_g(x)\log(1 - D(x))]\mathrm{d}x \end{aligned}$$

只要让积分内部的公式最大化，整个公式就可以实现最大化。这样问题就转变为最大化下面的公式：

$$f(D(x)) = p_r(x)\log D(x) + p_g(x)\log(1 - D(x))$$

对它进行求导取极值，可以得到

$$f'(D(x)) = p_r(x) \times \frac{1}{D(x)} - p_g(x) \times \frac{1}{1 - D(x)}$$

令上面的式子为 0，可以得到结果

$$D^*(x) = \frac{p_r(x)}{p_r(x) + p_g(x)}$$

这就是理论上判别式的预测结果，如果一张图像在真实分布中出现的概率大而在生成分布中出现的概率小，那么最优的判别模型会认为它是真实图像，反之则认为不是真实图像。如果生成模型已经达到了完美的状态，也就是说对每一幅图像都有 $p_r(x) = p_g(x)$，那么 $D^*(x) = \frac{1}{2}$。

可以利用上面的结果，计算当生成模型达到完美状态时，损失函数的值。我们将 $D^*(x) = \frac{1}{2}$ 的结果代入，可以得到

$$\begin{aligned} &= \int_x p_r(x)\log\frac{1}{2}\mathrm{d}x + \int_x p_g(x)\log(1 - \frac{1}{2})\mathrm{d}x \\ &= -\log 2 \int_x p_r(x)\mathrm{d}x - \log 2 \int_x p_g(x)\mathrm{d}x \\ &= -2\log 2 \end{aligned}$$

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

也就是说，生成模型损失函数的理论最小值为 $-2\log 2$。那么一般情况下，它的损失函数是什么样子呢？假设在某一时刻判别式经过优化已经达到最优，所以 $D^*(x)=\frac{p_r(x)}{p_r(x)+p_g(x)}$，将这个公式代入之前的公式，可以得到

$$
\begin{aligned}
&=\int_x p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}\mathrm{d}x+\int_x p_g(x)\log(1-\frac{p_r(x)}{p_r(x)+p_g(x)})\mathrm{d}x\\
&=\int_x p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}\mathrm{d}x+\int_x p_g(x)\log(\frac{p_g(x)}{p_r(x)+p_g(x)})\mathrm{d}x\\
&=\int_x p_r(x)\log(\frac{p_r(x)}{\frac{p_r(x)+p_g(x)}{2}}\times\frac{1}{2})\mathrm{d}x+\int_x p_g(x)\log(\frac{p_g(x)}{\frac{p_r(x)+p_g(x)}{2}}\times\frac{1}{2})\mathrm{d}x\\
&=\int_x p_r(x)[\log\frac{p_r(x)}{\frac{p_r(x)+p_g(x)}{2}}+\log\frac{1}{2}]\mathrm{d}x+\int_x p_g(x)[\log\frac{p_g(x)}{\frac{p_r(x)+p_g(x)}{2}}+\log\frac{1}{2}]\mathrm{d}x\\
&=-\log 2+\int_x p_r(x)[\log\frac{p_r(x)}{\frac{p_r(x)+p_g(x)}{2}}]\mathrm{d}x-\log 2+\int_x p_g(x)[\log\frac{p_g(x)}{\frac{p_r(x)+p_g(x)}{2}}]\mathrm{d}x\\
&=-2\log 2+\mathrm{KL}(p_r(x)\|\frac{p_r(x)+p_g(x)}{2})+\mathrm{KL}(p_g(x)\|\frac{p_r(x)+p_g(x)}{2})
\end{aligned}
$$

后面的两个 KL 散度的计算公式可以转化为 Jenson-Shannon 散度，简称 JS 散度，于是公式变为

$$=-2\log 2+2\times\mathrm{JSD}(p_{\mathrm{data}}\|p_g)$$

JS 散度与 KL 散度类似，都被用于衡量概率分布之间的距离。不同的是，JS 是一个对称的度量，也就是说下面的公式成立：

$$\mathrm{JS}(p\|q)=\mathrm{KL}(p\|\frac{p+q}{2})+\mathrm{KL}(q\|\frac{p+q}{2})=\mathrm{JS}(q\|p)$$

对称能带来什么好处呢？它能让散度度量更准确。接下来将用一段代码展示这其中的道理。首先给出两个离散随机变量的 KL 散度和 JS 散度的计算方法：

```
import numpy as np
import math
def KL(p, q):
    # p,q为两个list，里面存着对应的取值的概率，整个list相加为1
    if 0 in q:
        raise ValueError
    return sum(_p * math.log(_p/_q) for (_p,_q) in zip(p, q) if _p != 0)

def JS(p, q):
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    M = [0.5 * (_p + _q) for (_p, _q) in zip(p, q)]
    return 0.5 * (KL(p, M) + KL(q, M))
```

下面将用 3 组实验看看两个散度的计算结果，实现的目标是两个比较简单的离散分布。首先是第一个实验，我们直接求出两个分布在 KL 散度和 JS 散度下的距离：

```
def exp(a, b):
    a = np.array(a, dtype=np.float32)
    b = np.array(b, dtype=np.float32)
    a /= a.sum()
    b /= b.sum()
    print a
    print b
    print KL(a,b)
    print JS(a,b)

# exp 1
exp([1,2,3,4,5],[5,4,3,2,1])
```

它的结果如下：

```
[ 0.06666667  0.13333334  0.2         0.26666668  0.33333334]
[ 0.33333334  0.26666668  0.2         0.13333334  0.06666667]
0.521603083596
0.119687588569
```

在第二个实验中，我们将第二个分布稍做修改，将这个分布中的某个概率减小，这样两个分布的散度值将会增加，对应的代码如下所示：

```
# exp 2
exp([1,2,3,4,5],[1e-12,4,3,2,1])
exp([1,2,3,4,5],[5,4,3,2,1e-12])
```

它的结果如下所示：

```
[ 0.06666667  0.13333334  0.2         0.26666668  0.33333334]
[  9.99999982e-14   4.00000006e-01   3.00000012e-01   2.00000003e-01
1.00000001e-01]
2.06550201846
0.0985487692551
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
[ 0.06666667  0.13333334  0.2         0.26666668  0.33333334]
[  3.57142866e-01   2.85714298e-01   2.14285716e-01   1.42857149e-01
7.14285740e-14]
9.66295084712
0.193995300084
```

可以看出 KL 散度的波动比较大，而 JS 的波动相对小。

在第三个实验中，我们将修改第一个分布，同样是将其中一个概率值降低，代码如下所示：

```
# exp 3
exp([1e-12,2,3,4,5],[5,4,3,2,1])
exp([1,2,3,4,1e-12],[5,4,3,2,1])
```

它的结果如下所示：

```
[  7.14285740e-14   1.42857149e-01   2.14285716e-01   2.85714298e-01
3.57142866e-01]
[ 0.33333334  0.26666668  0.2         0.13333334  0.06666667]
0.742813156012
0.193995300084
[  1.00000001e-01   2.00000003e-01   3.00000012e-01   4.00000006e-01
9.99999982e-14]
[ 0.33333334  0.26666668  0.2         0.13333334  0.06666667]
0.383150755744
0.0985487692551
```

如果将第二个实验和第三个实验做对比，可以发现 KL 散度在衡量两个分布的差异时具有很大的不对称性。当第二个分布在某一个值上缺失时，KL 散度值就会发生很大的变化；而第一个分布在某一个值上缺失时，KL 散度并没有太大的波动。从这个例子可以很清楚地看出 KL 不对称性带来的问题。因为 JS 散度具有对称性，所以第二个实验和第三个实验中 JS 散度实际上是相同的。

从这个小例子可以看出，有时 KL 散度下降的程度和两个分布靠近的程度不成比例，而 JS 散度靠近的程度更令人满意，这也是 GAN 模型的一大优势。

15.4 GAIL 实现

15.3 节介绍了 GAN 的基本知识，让我们来看看 GAIL 的具体实现。经过前面章节

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

的介绍，我们知道回报函数的目标是使专家轨迹的长期回报尽可能地高于其他策略生成的轨迹。如果把这个目标放到 GAN 的场景下，就可以做一个对应。在 GAN 中，我们希望生成分布尽可能靠近真实分布；而在反向强化学习中，我们希望策略模型轨迹尽可能靠近专家轨迹。所以我们可以使用类似的方法，由策略模型充当 GAN 中的生成模型，以状态为输入生成行动；而回报函数模型可以充当判别模型，用于判别行动近似专家行动的程度。

令回报模型为 D，其参数为 w，策略模型为 π，其参数为 θ，这样我们的目标函数就变为

$$E_{\pi}[\log D_w(\boldsymbol{s},\boldsymbol{a})] + E_{\pi_E}[\log(1 - D_w(\boldsymbol{s},\boldsymbol{a}))] - \lambda H(\pi)$$

对于策略模型，我们希望最大化目标函数；对于回报模型，我们希望最小化目标函数。除了前面提到的 GAN 模型外，我们还可以把策略的熵添加进来，以保证策略分布相对平均。当我们对 GAN 有了一定的了解后，再来看 GAIL 的实现就变得容易理解了，对应的算法如下所示。

算法 Generative Adversarial Imitation Learning

Input

专家交互轨迹：$\boldsymbol{\tau}_E \sim \pi_E$；初始化策略和评判模型的参数：$\theta_0, w_0$

Start

for i = 0, 1, 2, ...**do**

使用策略 π_{θ_i} 与环境交互得到轨迹 $\boldsymbol{\tau}_i$

使用下面的梯度计算公式将评判模型的参数由 w_i 更新到 w_{i+1}：

$\hat{E}_{\boldsymbol{\tau}_i}[\nabla_w \log(D_w(\boldsymbol{s},\boldsymbol{a}))] + \hat{E}_{\boldsymbol{\tau}_E}[\nabla_w \log(1 - D_w(\boldsymbol{s},\boldsymbol{a}))]$

使用 TRPO 算法将策略模型的参数由 θ_i 更新到 θ_{i+1}，其中梯度为：

$\hat{E}_{\boldsymbol{\tau}_i}[\nabla_\theta \log \pi_\theta(\boldsymbol{a}|\boldsymbol{s}) Q(\boldsymbol{s},\boldsymbol{a})] - \lambda \nabla_\theta H(\pi_\theta)$

其中 $Q(\bar{\boldsymbol{s}},\bar{\boldsymbol{a}}) = \hat{E}_{\boldsymbol{\tau}_i}[\log(D_{w_{i+1}}(\boldsymbol{s},\boldsymbol{a}))|\boldsymbol{s}_0 = \bar{\boldsymbol{s}}, \boldsymbol{a}_0 = \bar{\boldsymbol{a}}]$

GAIL 的实现代码在 baselines/gail 文件夹中，其中包含了第 1 章提到的 Behavior Cloning 的实现和 GAIL 的实现，这里我们只关注 GAIL 的实现。

其中的判别模型由 adversary.py 文件中的 TransitionClassifier 实现，模型的输入为状态观测值和行动，对观测值进行标准化后，将其传入三层全连接层网络中进行计算，得到对行动判断的 logit 值。判别模型的目标函数有三部分：最大化来自专家轨迹的行动概率、最小化来自 Agent 策略轨迹的行动概率和最大化判别模型的熵。完整的计算图如图 15-7 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

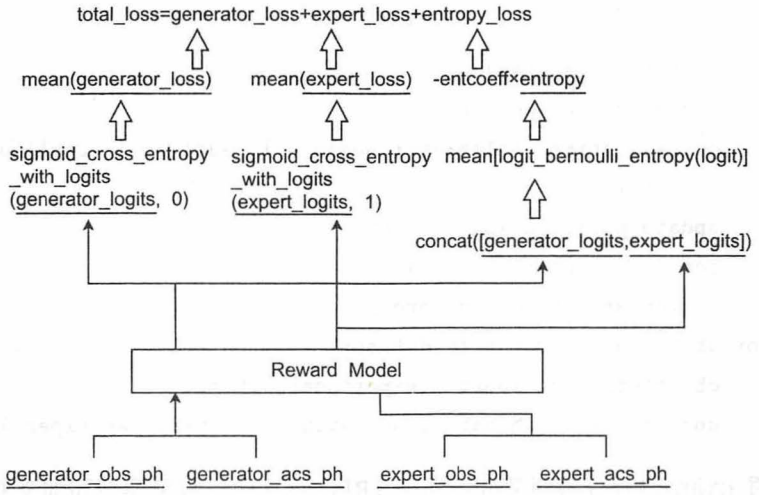


图 15-7 判别模型计算图

其中求解伯努利分布的熵的公式相对晦涩，笔者在此推导一下，伯努利分布的熵的公式为

$$H = -(1-p)\log(1-p) - p\log p$$

而要想得到伯努利分布的概率，需要用 sigmoid 函数对神经网络中得到的 logit 值做变换：

$$p = \frac{1}{1+\mathrm{e}^{-x}}$$

综合两个公式，可以得到

$$\begin{aligned}
H &= -(1-\frac{1}{1+\mathrm{e}^{-x}})\log(1-\frac{1}{1+\mathrm{e}^{-x}}) - \frac{1}{1+\mathrm{e}^{-x}}\log\frac{1}{1+\mathrm{e}^{-x}} \\
&= -(1-\frac{1}{1+\mathrm{e}^{-x}})\log(\frac{\mathrm{e}^{-x}}{1+\mathrm{e}^{-x}}) + \frac{1}{1+\mathrm{e}^{-x}}\log(1+\mathrm{e}^{-x}) \\
&= -(1-\frac{1}{1+\mathrm{e}^{-x}})[-x-\log(1+\mathrm{e}^{-x})] + \frac{1}{1+\mathrm{e}^{-x}}\log(1+\mathrm{e}^{-x}) \\
&= (1-\frac{1}{1+\mathrm{e}^{-x}})x + (1-\frac{1}{1+\mathrm{e}^{-x}})\log(1+\mathrm{e}^{-x}) + \frac{1}{1+\mathrm{e}^{-x}}\log(1+\mathrm{e}^{-x}) \\
&= (1-\frac{1}{1+\mathrm{e}^{-x}})x + \log(1+\mathrm{e}^{-x})
\end{aligned}$$

GAIL 中的策略模型训练算法选择了 TRPO 算法，代码结构和 TRPO 项目的结构比较类似。完整的模型训练如下面的伪代码所示：

```
def learn():
```


书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    while True:
        for i in range(g_step):
            seg = gen_seg(env, policy)
            ob, ac, atarg, tdlamret = seg['ob'], seg['ac'], seg['adv'],
                seg['tdlamret']
            update_policy_using_trpo(ob, ac, atarg)
            for j in range(vf_iters):
                update_vf(ob, tdlamret)
        for ob_batch, ac_batch in dataset:
            ob_expert, ac_expert = expert_dataset.get()
            update_critic(ob_batch, ac_batch, ob_expert, ac_expert)
```

可以看出训练过程分为两部分：训练 TRPO 算法中的策略模型和训练 GAIL 中的判别模型。策略模型同 Actor Critic 算法的模型结构相同，对于输入的状态，可以同时得到行动的概率分布和价值估计，因此这两部分都需要参与目标函数的计算，并进行优化，所以在代码的 6~8 行要完成两部分的优化，在代码的 9~11 行才进行判别模型的优化，其计算方法需要用到交互轨迹和专家轨迹两部分的数据。

如果读者对 TRPO 有一定的了解，就不必痛苦地了解 GAIL 代码的逻辑，其中更多的细节请读者自行阅读。

15.5 总结

本章介绍了两个反向强化学习算法，总结如下。

（1）最大熵反向强化学习以最大化序列概率分布熵为目标，同时限定基于策略模型的特征期望与最优策略的特征期望相等进行求解。

（2）生成对抗模仿学习基于生成对抗模型，构建了代表生成模型的策略模型和代表判别模型的回报模型，并通过交替训练策略模型和回报模型进行学习。

15.6 参考资料

[1] Ziebart B D, Maas A, Bagnell J A, et al. Maximum entropy inverse reinforcement learning[C]// National Conference on Artificial Intelligence. AAAI Press, 2008:1433-1438.

[2] Wulfmeier M, Ondruska P, Posner I. Maximum Entropy Deep Inverse Reinforcement Learning[J]. 2015.

书籍是人类进步的阶梯

[3] Ho J, Ermon S. Generative Adversarial Imitation Learning[J]. 2016.

[4] Goodfellow I J, Pouget-Abadie J, Mirza M, et al. Generative Adversarial Networks[J]. Advances in Neural Information Processing Systems, 2014, 3:2672-2680.

好书分享



《深度学习轻松学：核心算法与视觉实践》

作 者：冯超

出版时间：2017-08

ISBN：9787121317132

专注深度学习的思想本质，结合大量案例夯实技术基础！

知其然，更知其所以然，成为人工智能专家！

本书特色

√理论 + 实践 = 深度学习前沿技术

√夯实入门深度学习必备数学知识（线性代数基础、对称矩阵的性质、概率论、信息论基础、KL 散度、凸函数等）

√梳理深层模型（全连接层、卷积层）的基本知识

√经典深度学习框架入门

√深层模型的数值问题

√经典模型能完成哪些任务

√模型的优化算法（方法）

√视觉领域前沿应用（图像分割、生成模型等）

Broadview® 博文视点·IT出版旗舰品牌
www.broadview.com.cn
技术凝聚实力·专业创新出版

近年来强化学习在学术界和工业界都受到了极大关注，很多工业界的巨头都在不断探索强化学习的实际应用。滴滴出行作为全球最大的移动出行公司，已经把强化学习应用到多个业务场景中，帮助用户和司机解决出行问题，构建城市智慧大脑。本书介绍了强化学习的经典算法及近年来发展中的一些卓越成果，同时将理论联系实际，深入分析了这些算法的具体实现。作为一本内容丰富的强化学习科普书籍，希望无论是强化学习领域的初学者还是有经验的研发人员，均可从书中得到收获。

—— 王征，滴滴出行 AI Labs 时空数据组负责人

本书的特色之一是紧跟强化学习技术发展的前沿，从基于值函数的算法、基于策略梯度的算法，一直介绍到生成对抗模仿学习算法，内容涵盖了近年来深度强化学习研究的众多突破性算法；特色之二是结合开源的深度学习框架 TensorFlow、强化学习仿真框架 Gym 及在这些框架上实现的算法集合 Baselines，针对性地讲解了核心算法的代码实现，出色地将算法理论和实践结合起来。对强化学习领域的科研人员而言，这是一本不可多得的优秀读物。

—— 章宗长，苏州大学副教授

机器学习特别是强化学习，是近年和将来的学术研究重点，也是业界热点问题。本书作者通过对强化学习的概述和用 Python 实现的具体实例的讲解，为读者指明了一条通过编程理解和实践强化学习算法的道路。本书适合刚开始接触强化学习的初学者入门或者有经验的从业者巩固和借鉴。

—— 李可，美国微软公司高级数据科学家

这几年，机器学习因为深度学习的突飞猛进有了激动人心的进展，强化学习技术由于建模和环境的互动及延迟的反馈获得了较高的学术地位。本书全面而实用，对强化学习感兴趣的朋友来说是难得的教材。

—— 吴双，依图科技研究科学家

作者简介



冯超，毕业于中国科学院大学，滴滴出行 AI Labs 时空数据组专家算法工程师，曾任小猿搜题算法负责人之一。自 2016 年起在知乎开设技术专栏《无痛的机器学习》，发表与深度学习和强化学习相关的文章，文章以轻松幽默的语言、细致深入的分析为特点，得到了广泛的关注。曾撰写深度学习进阶领域口碑技术书《深度学习轻松学：核心算法与视觉实践》。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

上架建议：计算机>人工智能

ISBN 978-7-121-34000-0

定价：80.00元

责任编辑：郑柳洁
封面设计：吴海燕

欢迎投稿
邮箱：zhenglj@phei.com.cn
微信号：Alinamercy